Corosol : une machine virtuelle Java dynamiquement adaptative

Christophe Deleray

UNIVERSITÉ DE MARNE-LA-VALLÉE

Année :
2006

THÈSE

pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE MARNE-LA-VALLÉE

Discipline : **INFORMATIQUE**

présentée et soutenue publiquement

par

Christophe Deleray

le 18 octobre 2006

Titre :

Corosol : une machine virtuelle Java dynamiquement adaptative.

Directeur de thèse :

Jean Berstel

JURY

M. Jean Berstel, *Directeur*
M. Kim Mens, *Rapporteur*
M. Lionel Seinturier, *Rapporteur*
Mme Laurence Duchien, *Examinatrice*
M. Nicolas Bedon, *Examinateur*
M. Gaël Thomas, *Examinateur*

Remerciements

Je remercie très sincèrement :

Jean Berstel de m'avoir permis d'effectuer cette thèse au sein de l'institut Gaspard Monge dans d'excellentes conditions de travail et d'encadrement.

Kim Mens et Lionel Seinturier pour m'avoir fait l'honneur d'être rapporteurs de ma thèse, mais également à Gaël Thomas et Laurence Duchien pour celui d'être membre de mon jury, malgré un emploi du temps très chargé en ce début d'année.

Nicolas Bedon pour m'avoir prodigué de nombreux conseils avisés et pertinents durant toute la réalisation de cette thèse. Ses nombreuses relectures et ses remarques constructives et patientes m'ont beaucoup aidées durant celle-ci.

Également les autres membres de l'institut Gaspard Monge avec qui j'ai pu collaborer, dont Rémi Forax pour son paquetage Java `JMMF` dont je me suis servi pour les tests de Corosol et ainsi que ces précieux conseils lors du début de l'implantation de celle-ci.

Je remercie particulièrement mes parents et ma sœur qui m'ont soutenus et encouragés durant toute la durée de mes travaux et dans les moments difficiles. Également mon *comité de soutien* venu lors de ma soutenance ou qui m'y ont accompagné en pensée, ainsi que tout le reste de la *Rosny team*. À tous ces proches va mon affection la plus profonde.

Résumé

De nombreux outils ont été développés pour faire évoluer au cours du temps une application et/ou sa plate-forme d'exécution en fonction de l'apparition de nouveaux besoins. Dans le cadre de Java, ils se présentent souvent sous la forme de nouveaux compilateurs manipulant un langage Java enrichi, de chargeurs de classes évolués qui modifient le code compilé de l'application afin d'ajouter à celle-ci de nouvelles propriétés, ou encore de machines virtuelles spécialisées. Cependant ces différents outils sont difficilement combinables entre eux et/ou réutilisables en dehors de leur contexte. C'est pourquoi nous proposons, dans le cadre de Java, une machine virtuelle ouverte et entièrement écrite en Java pour conserver la portabilité, capable d'évoluer au cours du temps.

Elle dispose d'une architecture à composants qui s'inspire le plus fidèlement possible de la spécification de Sun Microsystems. Un composant de Corosol est une abstraction des unités fonctionnelles d'une machine virtuelle (comme le chargeur de classes ou les fils d'exécution), mais aussi des unités de stockage (comme le tas ou la pile d'exécution) et de tous les mécanismes internes liés à l'exécution (comme la résolution dynamique de méthodes). L'architecture de Corosol est décrite par des interfaces, qui spécifient les services des composants mais aussi comment ils s'intègrent dans Corosol et communiquent entre eux. Une implantation par défaut est fournie pour chaque composant. L'implantation de chaque composant peut être choisie avant le démarrage de Corosol par simple paramétrage. Cependant, l'originalité principale de notre machine virtuelle est une puissante interface d'introspection. Elle permet d'une part à l'application de consulter à tout instant les caractéristiques de sa plate-forme d'exécution afin de pouvoir s'y adapter. D'autre part, elle autorise l'application qui s'exécute à créer des composants pour la machine virtuelle à partir d'objets qui lui sont propres, et à changer les composants existants par les nouveaux, à n'importe quel moment. Notre travail a principalement consisté à définir et implanter cette interface d'introspection.

Abstract

Many tools were developed to make evolve in the course of time an application and/or its execution platform according to the appearance of new needs. Within the framework of Java, they are often appeared as new compilers handling a Java language enriched, advanced class loaders which modify the bytecode of the application in order to add new properties to this one, or specialized virtual machines. However these various tools are not easily combinable between them and/or reusable apart from their context. This is why we propose, within the framework of Java, an opened virtual machine, entirely written as Java to preserve the portability, and able to evolve in the course of time.

It has an component architecture which is inspired most accurately possible by the specification of Sun Microsystems. A Corosol component is an abstraction of the functional units of the Java virtual machine (like the class loaders or the threads), but also of the storage units (like the heap or the thread stacks) and all intern mechanisms related to the execution (like the method dynamic resolution). The Corosol architecture is described by interfaces, which specify the component services but also how they are integrated in Corosol and communicate between them. A default implementation is provided for each component. The implementation of each component can be selected before the starting of Corosol by simple parameter setting. However, the principal originality of our virtual machine is this powerful interface of introspection. Thanks to it, during its own execution, the application can consult the characteristics of its platform of execution in order to adapt its virtual machine. It also authorizes the application to create components for our virtual machine and to change the existing components by new ones. Our work mainly consisted to define and implements this introspection interface.

Table des matières

Table des figures

Liste des tableaux

Chapitre 1

Introduction

1.1 Contexte

Java est devenu un des langages de développement les plus populaires au cours de ces dernières années. Cela est dû en grande partie à sa portabilité et à sa facilité d'apprentissage et aux nombreux outils fournis avec ou développés autour, qui permettent d'installer, de développer, de migrer une application rapidement et de la maintenir ou de l'étendre facilement.

Différents mécanismes offerts par Java allègent ainsi la tâche du programmeur. Dans la plupart des cas, ils se révèlent très pratiques et ont contribué au succès de Java. Il s'agit par exemple du recyclage automatique de la mémoire allouée pour les objets d'une application, le chargement dynamique des classes, des bibliothèques dédiées à la programmation concurrente ou encore de l'introspection. Cependant, dans d'autres cas, ces mécanismes se révèlent être des contraintes incontournables. Par exemple, le ramasse-miettes soulage le programmeur de la gestion de la mémoire, mais lui en interdit aussi une gestion fine. Le programmeur ne maîtrisant pas le déclenchement du ramasse-miettes, ce dernier peut rendre difficile la gestion de contraintes temporelles dans une application *temps-réel*. L'interface d'introspection de Java ne permet pas d'accéder, voire de modifier, les différents composants de sa plate-forme d'exécution, la *machine virtuelle Java*. Il est possible, cependant, de modifier et d'ajouter des chargeurs de classes, mais ce n'est pas le cas des algorithmes de ramasse-miettes ou d'ordonnancement des fils d'exécution.

D'une manière générale, le manque d'évolutivité du langage et la visibilité restreinte du programmeur sur la machine virtuelle posent des problèmes de différentes natures. Pour résoudre ces problèmes, il existe des outils qui permettent d'enrichir le code compilé avant son exécution par la machine virtuelle [CCK98, LG97, Chi00, Chi98, Dah01, BLC02]. De telles transforma-

tions permettent d'ajouter des méthodes ou encore des champs aux classes des objets de l'application. Le programmeur peut ainsi implanter de nouvelles propriétés transversales à l'application qu'il développe, sans modification la machine virtuelle Java. Il peut lui ajouter de nouvelles fonctionnalités en transformant son code initial (code source et/ou compilé). Il s'agit par exemple d'ajouter de nouvelles méthodes aux classes existentes ou de nouveaux champs ou encore d'en modifier le comportement.

Le plus souvent sous la forme de bibliothèques appelées *paquetages*[1], certains de ces mécanismes peuvent ou sont déjà intégrés dans les machines virtuelles Java. D'autres, même s'ils s'avèrent très utiles dans certaines situations, ne le sont pas pour des raisons de performances essentiellement. Par exemple, les situations d'interblocages ne sont pas détectées par la plupart des machines virtuelles, alors qu'elles pourraient l'être [VEBO01]. Il en est de même pour les situations de « course aux données » : il s'agit alors cette fois-ci de détecter ou d'éviter des accès concurrents aux objets [SBN+97]. Les solutions à cette catégorie de problèmes passent soit par des extensions du langage [SBN+97], soit par l'intégration des mécanismes nécessaires dans la machine virtuelle (comme, par exemple, dans la machine virtuelle Java JanosVM [THL01]), soit encore par des programmes analysant les traces de la machine virtuelle.

D'autres classes de problèmes sont liées au langage proprement dit. Leur résolution passe par une modification ou une extension du langage, qui peut être mise en oeuvre de différentes manières : par la modification du compilateur, par le développement dans le langage standard de paquetages fournissant la fonctionnalité attendue, par la modification de la machine virtuelle, ou par une combinaison de ces trois moyens. Par exemple, la *programmation par aspects* [BSL01] est un paradigme qui propose de structurer une application sur la base du concept des *aspects*, qui modularisent une préoccupation transverse à tous les modules de cette application. Il s'agit par exemple de la gestion des exceptions, de la persistance des données ou des communications distantes, mais aussi des invariants de données, des contrats ou encore des *design patterns* [GHJV99]. Des compilateurs étendus comme AspectJ [KHH+01] ou Handi-Wrap [BH02] proposent des mises en oeuvre des aspects par extension du langage. Le code des aspects, ainsi que les classes sur lesquelles ils portent est spécifié dans des classes spéciales. Le compilateur AspectJ mèle dans le *bytecode* des méthodes des classes qu'il produit le code fonctionnel des méthodes avec le code non fonctionnel fourni par les aspects. Les solutions comme AspectJ posent le problème de la non-connaissance, à la compilation, de toutes les classes pouvant intervenir pendant l'exécution de l'application :

[1] *packages*

il peut arriver que de nouvelles classes inconnues lors de la compilation, et sur lesquelles des aspects auraient été posés s'ils avaient été connus, surviennent lors de l'exécution. L'action des aspects sur les instances de ces classes n'est alors pas garantie. Des solutions à ce problème proposent des paquetages permettant de gérer dynamiquement les aspects [PDFS01, BH02]. Elles ne garantissent toutefois la pose correcte des aspects sur les objets concernés que si le paquetage est utilisé correctement, et allourdissent dans la plupart des cas le développement du programme et ses performances à l'exécution. Une pose dynamique des aspects ne peut être garantie que par une machine virtuelle spécifique. De plus, l'inclusion de mécanismes spécifiques dans la machine virtuelle améliore les performances du programme.

Il existe également des problèmes dont la solution passe par une modification du langage. Par exemple, le polymorphisme en Java est mis en oeuvre par son mécanisme d'appel de méthodes. Dans un appel de méthode Java, le choix de la méthode est fait sur le type de ses paramères et de l'objet cible au moment de la compilation, puis ce choix est prédis au moment de l'exécution par le type exact de l'objet cible. Certaines applications nécessitent pourtant qu'au moment de l'exécution le choix de la méthode se fasse sur le type de l'objet cible, mais aussi sur le type exact des arguments de l'appel. Par exemple, il est souvent nécessaire de fournir une implantation des méthodes `equals(Object)` ou `compareTo(Object)` aux objets d'une application Java. Le comportement de ces méthodes dépend non seulement du type dynamique de l'objet sur lequel elles s'appliquent, mais également du type dynamique de celui qui est spécifié en paramètre. Or, en Java, toute classe est sous-type de `Object`. Le programmeur doit ainsi explicitement déterminer quel est le type dynamique passé en paramètre au sein de son implantation. Il peut encore s'agir d'applications nécessitants des opérations de *glisser-déposer*[2]. Le résultat de telles actions dépend du type dynamique de l'objet déplacé et de celui dans lequel il est déposé. Encore une fois, ce problème peut être résolu par extension du compilateur, par l'ériture de paquetages [FDR00] ou par inclusion de la fonctionnalité dans la machine virtuelle.

Certains logiciels s'adaptent à leur environnement d'exécution. Il existe, ainsi, des logiciels qui modifient leur comportement en cours d'exécution en fonction de la variation de certains paramètres de leur plate-forme d'exécution pour des raisons d'optimisation. Ceci nécessite que la machine virtuelle dispose d'une interface proposant les fonctionnalités attendues par l'application.

D'autre part, certaines recherches récentes tendent à inclure des vérificateurs formels dans la machine virtuelle. Par exemple, il a été montré expéri-

[2] *drag-and-drop*

mentalement [BFGP02] qu'il n'était pas utopique d'utiliser Spin [Spi97] pour vérifier le bytecode d'une classe Java. Cette expérience a été faite par utilisation directe de Spin sur les fichiers contenant le bytecode des classes, mais la vérification du bytecode est normalement effectuée par la machine virtuelle au chargement de la classe, ce qui nécessite alors d'intégrer le vérificateur dans la machine virtuelle.

Les besoins de modifications de la machine virtuelle Java sont donc nombreux. Certains projets de recherche développent des machines virtuelles Java ayant des propriétés particulières. Citons par exemple KissMe [Tja99], qui implante la persistance, Jupiter [DA01a], dont l'implantation est orientée pour des machines parallèles, Jalapeño [BAS99b, BAS99a, BAW00] et JRockit [Aa02], spécialisées dans la haute performance des applications serveurs, et JanosVM [THL01], spécialisée dans la gestion de ressources pour les réseaux. D'autres proposent des machines virtuelles ouvertes, sans fonctionnalités particulières mais dont les composants sont modifiables. La plupart de ces dernières sont écrites en C (SableVM [GH01], Jupiter [DA01a]), certaines dans un mélange de C et de Java (Jalapeño [BAS99b, BAS99a, BAW00], Jikes RVM [Jik], KissMe [Tja99]) et d'autres, beaucoup plus rares, seulement en Java (JavaInJava [Tai98]). Les machines virtuelles érites en C sont plus performantes, mais leur modification impose au programmeur de connaître ce langage, et le prive des facilités de redéfinition propres à Java.

L'utilisation explicite de toutes ces machines virtuelles dans des programmes Java a l'inconvénient de la perte de portabilité des applications, qui est un des points forts de Java. D'autre part, leur modification éventuelle doit souvent être effectuée avant leur utilisation.

1.2 Objectifs

C'est dans ce contexte que se placent nos travaux de thèse. Nous présentons nos travaux de réalisation d'une machine virtuelle Java entièrement écrite en Java et adaptable aux besoins de l'application.

Le premier objectif est de proposer une architecture la plus modulaire possible pour une machine virtuelle Java. La définition des composants de cette machine virtuelle doit non seulement être fidèle à la spécification de Sun [LY99] mais aussi permettre de facilement remplacer ou ajouter un de ceux-ci avant son démarrage. Le choix du langage Java pour écrire la machine virtuelle permet d'en utiliser toutes les facilités de redéfinition ou d'extension des classes pour la redéfinition des composants. Aussi, toute modification apportée à notre machine sera portable, car elle sera exécutée par une autre machine virtuelle Java (qui elle est standard).

Le second objectif est d'offrir un mécanisme de réflexivité suffisamment puissant qui permet à l'application non seulement d'examiner son environnement d'exécution, c'est-à-dire les composants de notre machine virtuelle, mais surtout de pouvoir modifier celui-ci et donc la sémantique de sa propre exécution. L'application peut ainsi s'adapter à de nouveaux besoins en modifiant sa machine virtuelle.

Un des points importants de notre travail est la définition formelle d'une interface d'introspection totale et le découpage formel de la machine virtuelle en termes de composants et de leurs services. Cette définition formelle doit être correcte et typée, afin que chaque concept apparaissant dans la machine virtuelle soit clairement identifié.

Bien que nous nous attachons à ne pas obtenir une machine virtuelle aux performances catastrophiques, ce point n'est pas notre soucis principal : nous développons la machine virtuelle dans l'esprit d'obtenir une plateforme d'expérimentation de travaux de recherches en Java et non pour des applications industrielles.

1.3 Plan

Le document de thèse est organisé de la manière suivante :

1. La première partie dresse l'état de l'art des machines virtuelles adaptatives :
   - le chapitre 2 décrit les machines virtuelles utilisant la modularité de leur architecture et/ou la réflexivité pour adapter leur exécution.
2. La seconde partie présente le langage Java et sa machine virtuelle :
   - le chapitre 3 donne une brève présentation du langage Java, examine en particulier sa portabilité à l'exécutions, détaille la structure de son code compilé et montre comment celui-ci peut être simplement généré par modification de code existant ;
   - le chapitre 4 présente l'architecture de sa machine virtuelle.
3. La troisième partie présente Corosol, notre machine virtuelle Java :
   - le chapitre 5 fait le point sur les objectifs de nos travaux et décrit en particulier les différentes propriétés de Corosol ;
   - le chapitre 6 expose son architecture à composant et l'importance des mandataires dans celle-ci ;
   - le chapitre 7 se concentre sur son modèle d'exécution basé sur la réflexivité ;
   - le chapitre 8 détaille la construction des mandataires lors de l'exécution ; on montre en particulier comment le code compilé de ces derniers est généré.

4. La quatrième partie aborde des exemples d'adaptations de notre machine :
   - le chapitre 9 dresse un exemple d'adaptation avant l'exécution ; il montre comment supporter l'ajout d'un nouveau type primitif au langage Java.
   - le chapitre 10 présente plusieurs exemples d'adaptation de Corosol durant son exécution ; en particulier comment modifier le comportement d'une instruction de la machine virtuelle, comment remplacer le tas et l'ordonnanceur, et comment modifier l'algorithme de recherche de méthodes.
5. La dernière partie concluera cette thèse et exposera les perspectives de travaux futurs.

Première partie

Les machines virtuelles adaptatives

Chapitre 2

État de l'art des machines virtuelles adaptatives

Ce chapitre contient une présentation rapide d'une sélection de machines virtuelles proches de nos préoccupations. Il s'agit de machines virtuelles adaptatives, de machines virtuelles Java disposant de fonctionnalités particulières ou dont l'implantation présente des caractérisques architecturales ou conceptuelles intéressantes pour notre propos, comme par exemple le découpage de l'exécution en plusieurs niveaux.

2.1 Introduction

Le principe d'une machine virtuelle est de reproduire le comportement d'un processeur virtuel *via* l'exécution d'un programme sur un processeur physique réel. Elle est utilisée comme environnement d'exécution d'un langage portable de haut niveau. Ainsi, en pratique, la machine virtuelle exécute un programme qui lui est destiné, tandis qu'elle même est exécutée par un processeur physique (généralement *via* un système d'exploitation). En général, elle dispose d'un interprète du code produit pour le processeur virtuel et gère son propre environnement d'exécution. Les éléments de son architecture interne sont donnés par une spécification qui décrit les tâches qu'elle doit accomplir. Le code exécuté par une machine virtuelle Java est appelé *bytecode*.

En plus de faire évoluer l'application exécutée, une machine virtuelle *adaptative* peut évoluer au cours du temps. De nouvelles propriétés peuvent lui être ainsi ajoutées ou supprimées. Il existe deux moments pour réaliser ces opérations :

– avant l'exécution, ce qui correspond à une adaptation «statique»,

– pendant l'exécution, ce qui correspond à une adaptation «dynamique».

L'adaptation statique d'une machine virtuelle peut être tout d'abord réalisée par la réécriture entière de machines virtuelles *ad-hoc*. Citons par exemple Plava [Tja99] et OFL [CCL99], qui implante la persistance, JRockit [Aa02], spécialisée dans la haute performance des applications serveurs, ou encore JanosVM [THL01], spécialisée dans la gestion de ressources pour les réseaux. L'adaptation statique d'une machine virtuelle peut également reposer sur l'écriture de composants *ad-hoc* de la machine virtuelle : on s'appuie alors sur la modularité de l'architecture de la machine virtuelle.

L'adaptation dynamique d'une machine virtuelle peut être réalisée par l'utilisation de la réflexivité, qui peut rendre visible les éléments de son implantation à une application, comme pour la machine virtuelle du langage Smalltalk [GR83]. Des techniques corollaires sont également utilisées : des machines virtuelles telles que MetaXa [GK98] et Guaraná [OB99, OB98, OGB98] fournissent à l'application des objets particuliers, appelés *méta-objets* pour contrôler et/ou modifier le déroulement de leur exécution.

Commençons par la description des machines virtuelles adaptatives statiquement, avant l'exécution. Nous aborderons ensuite la cas des machines virtuelles dynamiquement adaptative.

2.2 Adaptation statique de l'exécution

Nous présentons les machines virtuelles statiquement adaptatives. Plutôt que d'aborder les nombreuses machines virtuelles Java *ad-hoc*, nous nous concentrons en particulier sur les machines permettant la redéfinition de certains composants de leur architecture.

2.2.1 JavaInJava

Le projet JavaInJava [Tai98] a débuté en octobre 1997 et s'est déroulé jusqu'en décembre de la même année. L'objectif était de déterminer si le langage Java était adapté pour la construction d'une machine virtuelle Java, en comparaison avec des langages comme C ou C++, habituellement utilisés. Cela a permis aussi d'étudier quelques difficultés de construction avec Java. JavaInJava est donc une machine virtuelle Java entièrement écrite en Java et interprétée par une autre machine virtuelle Java.

JavaInJava a aussi été développé pour être une machine virtuelle Java de référence. Un effort a été fait sur son implantation pour qu'elle soit extensible, pour servir de base à d'autres implantations de machines virtuelles.

Cette implantation a été voulue simple à comprendre en comparaison à des machines réalisées en C ou en C++. Dans la réalisation, les concepteurs se sont concentrés sur la clarté plus que sur les performances.

L'architecture de JavaInJava est donc orientée objet. Les auteurs ont modélisé jusqu'aux plus petites parties de la structure d'une JVM, comme l'ordonnanceur ou les *frames*. Toutes ces structures sont accédées *via* une unique classe : la classe `VM`, qui joue le rôle de façade par rapport aux autres petites structures. Elle cache l'implantation des composants internes de JavaInJava. De plus, il n'existe pas de variables globales (encore moins de champs publiques) accessibles à tous ces objets : tout est encapsulé dans des classes (aucun registre n'est utilisé, d'ailleurs Java ne le permet pas). JavaInJava représente en interne les champs des objets utilisateurs (c'est-à-dire créés par l'application) sous la forme d'objets. La valeur d'un type primitif est représentée par un objet encapsulant celle-ci, d'où des problèmes de représentation dans les *frames* et lors de l'initialisation (mais aussi de coercition (*cast*) ou de vérification de type). Il n'existe pas non plus de gestionnaire de mémoire, c'est-à-dire que JavaInJava ne dispose ni d'allocateur de mémoire (comparable au tas), ni de ramasse-miettes. En fait, chacun des objets de l'application est créé au sein du tas de la machine virtuelle qui interprète JavaInJava.

Il est à noter que JavaInJava rencontre des difficultés lors de l'exécution d'applications graphiques nécessitant l'utilisation de fonctions natives. En termes de lignes de code, ce projet représente environ 10000 lignes de codes réparties sur 42 classes, mais celui-ci n'est malheureusement pas disponible publiquement. Elle est environ trois fois plus lente qu'une machine virtuelle Java standard (sur certains tests, environ 700 fois plus lent).

De façon naturelle, l'architecture modulaire en Java de cette machine virtuelle se prête à la redéfinition pour s'adapter aux besoins d'une application. Cependant, rien n'est prévu pour que cette adaptation se fasse durant l'exécution. La redéfinition ne peut s'effectuer qu'avant démarrage de la machine.

2.2.2 Jikes RVM

Suite à une étude de faisabilité d'une machine virtuelle entièrement écrite en Java débutée en décembre 1997 , IBM démarre le projet Jalapeño [BAS99b, BAS99a, BAW00]. L'objectif est alors de développer une machine virtuelle Java ouverte et performante, entièrement écrite en Java, pour que les programmeurs d'applications serveurs puissent l'adapter aux caractéristiques spécifiques de l'application et de l'architecture matérielle du serveur.

En octobre 2001, le projet Jalapeño devient *open source* sous le nom de Jikes RVM (Research Virtual Machine) [Jik]. Depuis, de nombreux travaux

de recherches sur les machines virtuelles ou sur Java utilisent Jikes RVM comme plate-forme de développement ou de tests.

Différentes caractéristiques des machines virtuelles, cruciales pour les applications serveurs peuvent être paramétrées dans Jikes : l'allocateur de mémoires, le ramasse-miettes, la politique de gestion des processus légers, le compilateur *juste à temps* (JIT), ou encore la répartition de la charge de travail sur les différents processeurs. Pour chacune des fonctionnalités de la machine virtuelle, le programmeur peut choisir, au moment du paramétrage, son implantation par défaut, une implantation qu'il fourni ou une implantation choisie dans une bibliothèque distribuée avec Jikes. Par exemple, une bibliothèque contenant une variété importante de ramasse-miettes est fournie avec Jikes. Un script de configuration permet au programmeur de spécifier l'implantation désirée en la nommant. Le script génère alors du code Java pour la machine avec l'implantation voulue.

Jikes n'est pas réellement un *interprète* Java : il compile toujours le *bytecode* à exécuter en code natif de la plate-forme d'accueil . La compilation du *bytecode* Java en code natif est l'idée centrale de Jikes et guide toute son architecture. En fait, Jikes contient trois compilateurs :

- le *compilateur de base* transforme le *bytecode* Java en code natif, instruction par instruction, en simulant le comportement d'une machine virtuelle implantant la spécification. Ce compilateur est simple et rapide mais le code produit est aussi lent que celui exécuté par un interprète ;
- le rôle du *compilateur optimisé* est la production de code efficace pour les méthodes utilisées intensivement : utilisé comme un compilateur *juste à temps*, il permet l'optimisation du temps d'exécution ;
- le *compilateur rapide* est un compromis entre les deux autres compilateurs : il sait réaliser rapidement de petites optimisations sur le code produit.

Dans un premier temps, le compilateur de base transforme rapidement le code en Java en code natif. Si ce dernier est souvent utilisé, il est retransformé, pour être optimisé, par un des deux autres compilateurs. Le *bytecode* applicatif est fourni aux compilateurs par le *chargeur de classes*, qui l'obtient à partir de ressources systèmes comme des fichiers ou des connexions réseau. Le code peut être recompilé en fonction de mesures sur son temps d'exécution.

Du fait de l'architecture centrée sur ces trois compilateurs, il est difficile de modifier certaines caractéristiques particulières de la machine virtuelle : elles peuvent être dispersées partout dans le compilateur. Ainsi, les modifications des fonctionnalités de la machine virtuelle deviennent délicates. Par exemple, il est difficile d'ajouter un système de typage de la pile. En effet, la pile n'est pas un composant de la machine virtuelle clairement modélisé :

l'implantation de ses caractéristiques physiques et fonctionnelles est dispersée dans les instructions de *bytecode* qui la manipulent. Ajouter du typage sur la pile revient donc à modifier toutes ces instructions, dont les implantations sont elles-mêmes réparties dans le compilateur. De même, le manque d'une modélisation claire en composants de la machine virtuelle ne permet pas d'augmenter facilement l'interface d'instrospection standard de Java pour qu'elle présente la machine virtuelle sous la forme de composants.

La portabilité n'est pas un élément essentiel du projet Jikes. En effet, pour obtenir des performances optimales (ce qui est un objectif primordial) il est important de pouvoir exploiter les caractéristiques physiques de la plate-forme d'accueil. Cependant, dans le but de faciliter l'utilisation sur différents types de plate-formes, Jikes possède un mécanisme d'auto-compilation. Le compilateur interne à Jikes est utilisé pour transformer le *bytecode* de Jikes en code natif de la plate-forme d'exécution : ainsi Jikes ne nécessite pas de machine virtuelle Java hôte pour pouvoir s'exécuter. L'exécution de Jikes sur une plate-forme particulière nécessite une phase de compilation préparatoire qui, à partir du *bytecode* de Jikes, calcule l'image mémoire d'un processus exécutable qui contient le code des différents services qu'offre la machine virtuelle, ainsi qu'une zone de *bootstrap* dont le rôle est de charger le *bytecode* applicatif, de le compiler et d'en démarrer l'exécution. Le code applicatif compilé peut directement utiliser le code de la machine virtuelle compilée : ils sont dans le même espace d'instructions. Bien que Jikes soit entièrement écrite en Java, ses performances sont similaires à celles d'autres machines virtuelles Java, comme celles écrites en C ou en C++.

2.2.3 Jupiter

Jupiter [DA01a] est une infrastructure extensible et modulaire d'une machine virtuelle Java conçue pour être exécutée sur un *cluster* de stations de travail, soit 128 processeurs en tout. Dans sa conception, Jupiter dispose d'une mémoire locale pour chaque processeur, afin d'assurer une plus grande rapidité d'accès que celle à la mémoire globale. Celle-ci est d'ailleurs soumise à un ramasse-miettes parallèle, par opposition à un ramasse-miettes n'utilisant qu'un seul processus léger. En effet les algorithmes «stop the world», qui arrêtent momentanément l'exécution de l'application pendant que le ramasse-miettes effectue son travail, ne sont pas appropriés pour les machines multi-processeurs, car le coup de l'arrêt de l'exécution impose plus de pénalité lors de l'exécution que pour une machine mono-processeur.

Bien qu'écrit en langage C, il s'agit aussi d'une machine virtuelle fortement modulaire et extensible. Dans sa conception, Jupiter a été construite comme un ensemble de petites unités, chacune accessible par une interface de

programmation. Ces unités ou modules ont été conçus pour être atomiques, c'est-à-dire qu'une modification de Jupiter ne devrait pas entraîner la césure d'un module en deux sous modules. Chacun des modules a aussi été réalisé pour qu'il y ait cohérence, c'est-à-dire qu'un changement de Jupiter ne devrait entraîner que le changement d'un très petit nombre de modules, voire que d'un seul. Enfin, ils ont été conçus pour être indépendants, c'est-à-dire que des modifications orthogonales de Jupiter ne devraient entraîner que la modification de modules distincts. Bien qu'entièrement développée en C, Jupiter privilégie une approche orientée objet dans sa réalisation. Chacun des modules possèdent une implantation de base qui peut être redéfinie. À chacun des modules est associée une interface. Elle est entièrement définie par un ensemble de méthodes et des structures de données et chacun des champs des modules est accédé *via* une fonction.

Ainsi, de part ce qui précède, Jupiter est capable de s'adapter à une application parallèle afin que son exécution soit distribuée sur un ensemble de processeurs. Bien que facilitée par des interfaces adaptées et orientée objets, la redéfinition du comportement de cette machine virtuelle n'a pas lieu durant son exécution, mais plutôt avant l'exécution de celle-ci. Le concepteur de l'application reprogramme chacun des modules qui conviennent après les avoir identifiés.

2.3 Adaptation dynamique de l'exécution

Nous présentons maintenant les machine virtuelles dynamiquement adaptatives. Cette propriété repose en particulier sur l'utilisation de mécanismes réflexifs. Avant d'aborder ces différentes machines, revenons sur les notions liées à ces mécanismes.

2.3.1 Notions de réflexivité

La **réflexivité** est la capacité d'un programme, pendant qu'il s'exécute, à manipuler son propre état d'exécution en tant que données. On distingue deux aspects d'une telle manipulation : l'**introspection** et l'**intercession**.

2.3.1.1 Introspection, intercession, réification

L'introspection est la capacité d'un programme d'observer et donc de raisonner sur son propre état d'exécution, c'est-à-dire de répondre à des interrogations sur celui-ci. En Java, on utilise par exemple la classe `java.lang.Class`

pour obtenir et interroger l'état d'une classe d'un programme lors de l'exécution. Il est alors possible d'obtenir la liste des méthodes de celle-ci ou encore la liste de ses constructeurs ou de ses variables d'instances.

L'intercession est la capacité d'un programme à pouvoir cette fois-ci modifier son propre état d'exécution ou encore altérer la sémantique de sa propre exécution, c'est-à-dire la façon dont il doit être compris ou interprété.

Chacun de ces deux mécanismes, introspection et intercession, requiert un encodage de l'état d'exécution d'un programme en tant que données, appelé **réification**.

2.3.1.2 Réflexivité structurelle et comportementale

On désigne par **système** le couple {programme, interprète}. Lorsqu'un système ne réifie que le programme exécuté ou bien les structures de données du langage utilisé, la réflexivité est dîte **structurelle**. Dans le cas d'un programme écrit dans un langage de classes comme Java ou Smalltalk, les données réifiées peuvent correspondre aux classes, aux champs ou au méthodes utilisées au sein du programme.

Lorsque, cette fois-ci, un système réifie les structures de données liées à son propre fonctionnement, c'est-à-dire celles de l'interprète, la réflexivité est dîte **comportementale**. Cela peut être, par exemple, réifier la pile d'exécution ou les mécanismes sous-jacents, pour les appels de méthodes. Dans le cas de l'intercession, la réflexivité comportementale a donc une incidence sur la sémantique de l'exécution du système.

2.3.1.3 Causalité

La modification d'un système réflexif a des conséquences sur son état interne. Cela a donc des répercussions sur la réification de ce dernier. Un système réflexif et sa représentation interne réifiée sont donc intimement liés : on dit qu'ils sont **liés causalement**, la modification de l'un entraînant la modification de l'autre.

2.3.1.4 Niveau de base et niveau méta

Les travaux de Brian Smith dans 3-Lisp [Smi82, Smi84] découpent l'exécution en plusieurs niveaux. Tout d'abord, on distingue le **niveau de base**, qui constitue le niveau où se situe le programme à exécuter. Ce niveau est contrôlé par un niveau supérieur, celui de l'interprète du programme qui est appelé le **niveau méta**.

La figure 2.1 illustre les notions de réflexivité, de réification, et leur organisation en niveaux :

- la création d'une représentation manipulable de l'état du système, c'est-à-dire la réification, est une action du niveau de base vers le niveau méta ; cette représentation appartient au niveau méta tandis que les éléments du système appartiennent au niveau de base.
- la représentation de son état construite au niveau méta et utilisée depuis le niveau de base, c'est-à-dire la réflexivité, provoque un passage du niveau méta vers le niveau de base ; grâce à cette représentation, le système peut répondre depuis le niveau de base à des interrogations sur son état ; il peut aussi contrôler sa propre exécution qui se déroule à ce niveau de base.

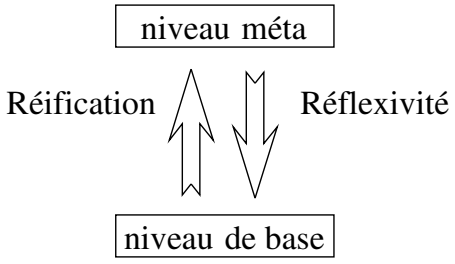

FIG. 2.1 – Réflexivité et réification

2.3.1.5 Protocole à Méta-Objets (MOP)

Les modifications de l'état du système se font par l'intermédiaire d'une interface de programmation nommée MOP (*Meta Object Protocol*) ou *protocole à méta-objets*. Elle permet de manipuler des objets particuliers appelés **méta-objets** et appartenant au niveau méta. Leur fonction est de contrôler l'exécution des objets du niveau de base.

Deux types de MOPs sont à distinguer [Zim96] :

- les MOPs *implicites*, utilisés de façon transparente au niveau de base et utilisés automatiquement pour contrôler l'exécution de celui-ci,
- les MOPs *explicites*, utilisés explicitement par le niveau de base, comme celui du paquetage Java `java.lang.reflect`, que le programmeur peut utiliser, mais dont la machine virtuelle ne se sert pas pour l'exécution d'une application.

2.3.1.6 Lien méta

Le lien qui unit un (ou plusieurs) méta-objets à un (ou plusieurs) objets du niveau de base est appelé **lien méta**.

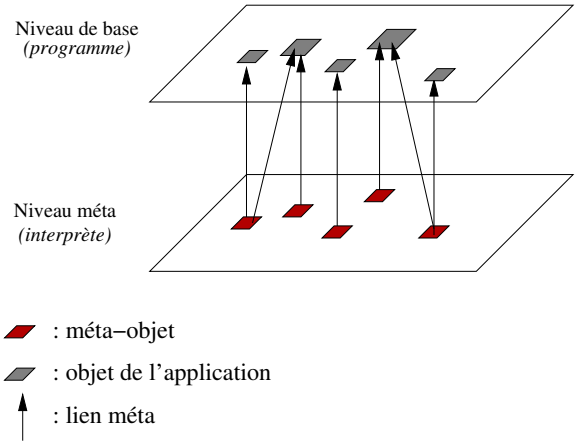

FIG. 2.2 – Niveau de base et niveau méta

2.3.2 Les machines virtuelles dynamiquement adaptatives

Nous avons présenté les notions liées à la réflexivité. Cela nous aidera à mieux comprendre le fonctionnement des machines virtuelles dynamiquement adaptatives que nous décrivons dans la suite.

2.3.2.1 Smalltalk

Smalltalk est un langage réflexif dont tous les éléments peuvent être réifiés, dont :

- les classes et leur méta-classes ;
- les méthodes ;
- le compilateur ;
- la pile d'exécution.

La machine virtuelle de Smalltalk est entièrement décrite en terme d'objets Smalltalk[GR83], accessibles et modifiables durant l'exécution. Elle manipule les classes de l'application, mais aussi celles qui appartiennent à sa propre implantation, et les rend accessibles lors de l'exécution. Elle gère également les méta-classes dont les classes sont des instances. Classes et méta-classes encapsulent le comportement des futures instances (en particulier la dispositions des champs). Ce modèle permet entre autres la redéfinition des classes lors de l'exécution, mais également de d'acceder à la structure de leurs instances. Il est par exemple possible de contourner l'encapsulation en accédant aux variables d'instances *via* des indices plutôt que par leur noms.

L'exemple ci-après, dans lequel (2@3) représente une instance d'une classe `Point`, montre comment la méthode `instVar` permet d'obtenir la valeur de la variable d'instance d'indice 1 :

```
(2@3) instVarAt: 1
```

Smalltalk réifie également la pile d'exécution de sa machine virtuelle composée de contextes d'exécution empilés pour chaque appel de méthode. Chaque contexte possède en particulier la méthode à exécuter, qui contient le code compilé à exécuté, également réifiable lors de l'exécution. Les arguments de la méthodes sont aussi présents dans chaque contexte : ils peuvent également être examinés.

Le processus de compilation de Smalltalk est incrémental. Les classes sont compilées au fur et à mesure. Il est ainsi possible d'ajouter de nouvelles méthodes et de nouveaux champs à un classe sans redémarrer la machine virtuelle pour que les modifications soient prises en compte. Tous les objets intervenants au cours de ce processus, en particulier, l'analyseur syntaxique, l'arbre qui en résulte mais également le générateur de code, sont réifiables. La sémantique de Smalltalk est peut être ainsi étendue à la volée (voir par exemple [Riv96]).

D'autres travaux ont utilisé les capacités réflexives du langage pour modifier d'autres mécanismes, comme les appels de méthodes [FJ89].

Squeak Le projet Squeak [IKM+97] a commencé en décembre 1995. Il s'agit d'un environnement de développement au sein duquel des logiciels éducatifs peuvent être utilisés et voire programmés par des non-initiés. Son objectif est également d'être utilisé sur des médias tels que des PDA ou l'Internet, ce qui implique pallier aux contraintes fortes dues à l'hétérogénéité des systèmes d'exploitation, des machines hôtes, des temps de chargement ou encore à la taille du code à exécuter.

Squeak consiste en une machine virtuelle Smalltalk entièrement écrite en Smalltalk et d'une image, c'est-à-dire un objet mémoire, contenant une bibliothèque de classes et l'état de la machine virtuelle composé de tous ses objets sous la forme d'un fichier. Pour atteindre de bonnes performances, Squeak dispose également d'un compilateur *juste-à-temps* qui traduit le code compilé Smalltalk en du code natif.

Squeak est ainsi un système entièrement réflexif et adaptatif, sa machine virtuelle étant ouverte : son code source peut être examiné et tout ce qui compose son environnement peut être modifié *via* les possibilités réflexives offertes par Smalltalk.

2.3.2.2 3-Lisp

Dans les années 80, les travaux de Brian Smith [Smi82, Smi84] ont permis de définir la notion de système réflexif. Smith est ainsi à l'origine de 3-Lisp, une extension du langage de programmation Lisp. Dans celui-ci, Smith effectue une distinction entre l'interprète qui exécute le programme et l'interprète exécutant le code réflexif.

3-Lisp définit donc plusieurs niveaux d'exécution. Tout d'abord le niveau de base qui constitue le programme à exécuter. Vient ensuite le méta-niveau correspondant à l'interprète du programme. Puis, en cas d'exécution de procédure réflexive au niveau de base, 3-Lisp provoque la création d'un nouveau méta niveau réflexif pour exécuter ce code. Ce niveau interprète le méta-niveau de l'interprète et est situé juste en dessous de l'interprète du niveau de base. Cependant, le code réflexif qui est exécuté peut être appelé récursivement. Ainsi, 3-Lisp considère un nombre potentiellement infini de méta-niveaux, chacun d'entre eux interprétant le méta-niveau qui se trouve juste au dessus, à l'exception du niveau de base (voir figure 2.3).

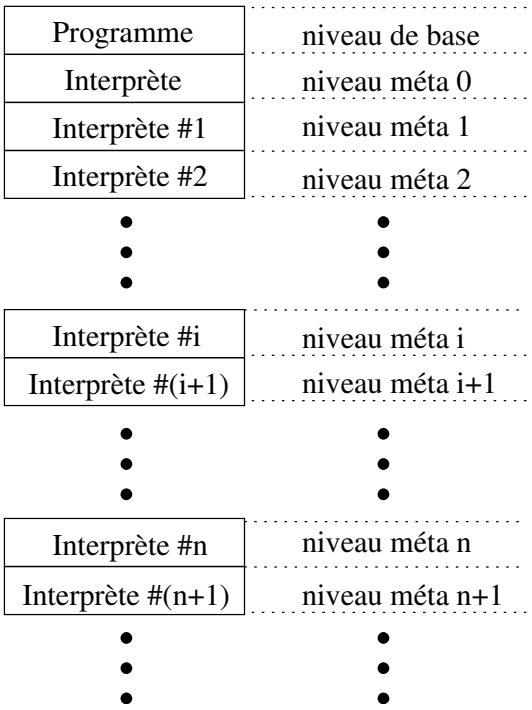

FIG. 2.3 – Les méta-niveaux

Le code de chacun des méta-niveaux est ainsi exécuté par un interprète qui contient son propre interprète issu du niveau inférieur. En modifiant ou

en construisant de tels éléments, une procédure réflexive peut ainsi contrôler le déroulement de l'exécution au niveau inférieur. Comme expliqué précédemment, la réflexivité pouvant être arbitrairement appelée par récursivité, 3-Lisp est aussi défini comme une tour infinie de processus 3-Lisp, chacun engendrant le processus immédiatement inférieur. Ainsi, dans 3-Lisp, le code réflexif est exécuté au même niveau que l'interprète et non par celui-ci, c'est-à-dire au niveau où se déroule actuellement l'exécution.

Chaque méta-niveau possède aussi son propre état de l'interprète, qui consiste en des structures d'environnement et de continuation. Chacun des niveaux, à l'exception du niveau de base, est donc en train d'exécuter le code d'un interprète réflexif. Les procédures réflexives à exécuter sont intégrées dans ce code.

Lors des appels de procédures réflexives, l'environnement d'exécution ainsi que l'endroit où doit se continuer l'exécution sont réifiés. Ils sont passés en arguments d'appels de cette procédure. Lors de l'appel, chaque procédure réflexive possède ainsi les arguments suivants :

- une structure représentant le code réflexif à exécuter ; par analogie avec le λ-calcul, ce code est appelé *redex*, car il s'agit d'une expression (*sous-terme*) qui doit être réduite (*normalisée*) par évaluation ;
- une structure représentant l'environnement de l'exécution avant normalisation du rédex réflexif ; c'est une structure décrivant l'environnement d'exécution de l'interprète en cours d'exécution ; celle-ci est liée causalement à l'état de l'interprète ;
- une structure de continuation prête pour accepter le résultat de la normalisation.

L'interprète de chaque procédure réflexive fait donc référence au contexte du programme exécuté au niveau inférieur, ce qui est aussi réalisé par les procédures réflexives. C'est ainsi que 3-Lisp réalise la réflexivité comportementale, qui est la capacité d'un programme d'observer et de modifier ses structures de données en train d'être utilisée pour la propre exécution de ce programme.

2.3.2.3 Virtual Virtual Machine (VVM)

Le projet Virtual Virtual Machine (VVM) [FPR98, PFSB00, Fol00, FBPS] a pour objectif d'offrir un environnement d'exécution générique (dit *virtuel*) et sécurisé pour n'importe quelle application *bytecodée* nécessitant une machine virtuelle. Cet environnement d'exécution est multi-langages et indépendant du matériel. Il est dynamiquement extensible. Il se veut aussi adaptable à un domaine donné, comme par exemple les cartes à puces, les téléphones

mobiles, les ordinateurs personnels ou encore les satellites. Il est aussi être exensible par modifications à la volée : de nouvelles fonctionnalités ou de nouveaux algorithmes peuvent être ainsi apportées. Il assure également l'interopérabilité entre les applications (par exemple en ce qui concerne l'échange de données).

Il s'agit donc d'un environnement d'exécution spécialisable, extensible et reconfigurable. Il est basé sur une machine virtuelle supportant l'exécution de machines virtuelles (elles-mêmes supportant l'exécution d'applications *bytecodées*). Cette machine virtuelle virtuelle (MVV) s'interfaçe entre un système d'exploitation existant (ou à venir) et des applications écrites dans plusieurs langages *bytecodés* (Java, Smalltalk, Caml, Lisp...).

La MVV est constituée d'un **processeur virtuel** qui est un moteur d'exécution de bas niveau. L'architecture de son modèle de mémoire et celle de son modèle d'exécution sont tous les deux simples. Ce processeur virtuel possède aussi un jeu d'instructions élémentaires. L'objectif est de fournir le support minimal, nécessaire à la construction de différents interprètes de *bytecode* (pas forcément Java, mais n'importe quel autre code de ce type, nécéssitant d'être interprété). Le *bytecode* d'une application est ainsi dynamiquement traduit en termes d'instructions pour le processeur virtuel, par génération de code à la volée. On fait concorder chaque instruction avec une unique instruction du processeur virtuel. Ce peut être aussi une nouvelle instruction ; elle est alors construite à partir des instructions de base du processeur virtuel. L'interopérabilité interne (c'est-à-dire la réutilisation des composants écrits dans différents langages) est facilitée par la traduction en cette unique représentation exécutable, mais aussi par le caractère réversible de celle-ci. Il n'existe donc qu'un seul mécanisme d'exécution pour lancer une application.

La MVV dispose aussi d'un **système d'exploitation virtuel**. Il est une abstraction des primitives d'un système d'exploitation natif, comme le système de fichiers, l'ordonnancement, l'accès aux ressources protégées ou les communications. Il se situe au-dessus de celui-ci. Il gère l'allocation et les ressources dont a besoin la MVV. Les services de ce système sont accédés *via* les primitives du processeurs virtuel. Il fournit ainsi les primitives d'accès aux ressources de la machine à chaque machine virtuelle décrit par ce qui est appelé une **VMLet**.

Une `VMLet` type une application et décrit un modèle d'exécution. Elle correspond ainsi à une description d'une machine virtuelle. Elle décrit en particulier :

- les instructions de cette machine virtuelle et comment elles sont transformées en termes d'instructions pour le processeur virtuel,
- le chargeur de *bytecode* de celle-ci,
- les règles de vérification de ce *bytecode*,

- le modèle objet de cette machine virtuelle et comment il se traduit en celui de la MVV,
- les primitives systèmes accédées par les instructions de celle-ci en termes de primitives du système d'exploitation virtuel,
- les règles de sécurité au niveau de l'exécution.

Une `VMLet` est chargée à la demande lorsqu'un nouveau type d'application survient. La MVV se spécialise ainsi par l'intermédiaire de ce modèle d'exécution chargé à la volée. Chaque `VMLet` ajoutant de nouvelles primitives à l'interprète de VVM, l'interopérabilité est donc assurée : VVM peut ainsi exécutée des applications provenant de domaines d'exécution différents. Il n'existe donc qu'un seul niveau d'interprétation, d'où de meilleures performances.

Chaque `VMLet` est écrite dans un langage de haut niveau et encodée dans un *bytecode*. Elles ne sont pas seulement déclaratives mais aussi impératives, car elles peuvent modifier leur propre environnement d'exécution. Ainsi, la MVV est une machine virtuelle réflexive exécutant des `VMLets`, ces dernières étant chargées de transformer celle-ci selon la description emportée par chacune d'elles.

La MVV dispose aussi d'un **module de sécurité** qui vérifie l'intégrité du code de l'application (relativement à la description donnée par la `VMLet`) ainsi que le partage des objets entre les langages.

La figure 2.4 montre les différents éléments de l'architecture de la MVV. Le processeur virtuel (*interpreter*) et le système d'exploitation virtuel (*minimal ressource access*) y sont représentés. Cette figure illustre la génération de code à la volée effectuée lors du chargement d'une `VMLet`.

2.3.2.4 MetaXa

MetaXa [GK96, GK98, Mic97] est un système réflexif écrit dans le cadre de Java. Ce projet a pour but de permettre la réflexivité structurelle, mais aussi la réflexivité comportementale, en particulier au niveau de l'invocation des méthodes. Plutôt que de choisir une approche basée sur la modification du langage de programmation ou du *bytecode*, MetaXa choisi d'étendre la machine virtuelle Java en lui ajoutant un module de création de méta-objets et un service d'événements destiné à intercepter leur utilisation. MetaXa est ainsi constitué d'une collection de méthodes natives écrites en C et regroupées au sein d'une DLL (*Dynamic Link Library*).

Le service d'événements permet de faire communiquer le niveau de base et le niveau méta. Un événement est créé au niveau de base et est propagé au niveau méta. Ainsi, lors de la création d'un objet, lors d'une invocation de méthode, lors du chargement d'une classe ou encore lors de l'accès à une

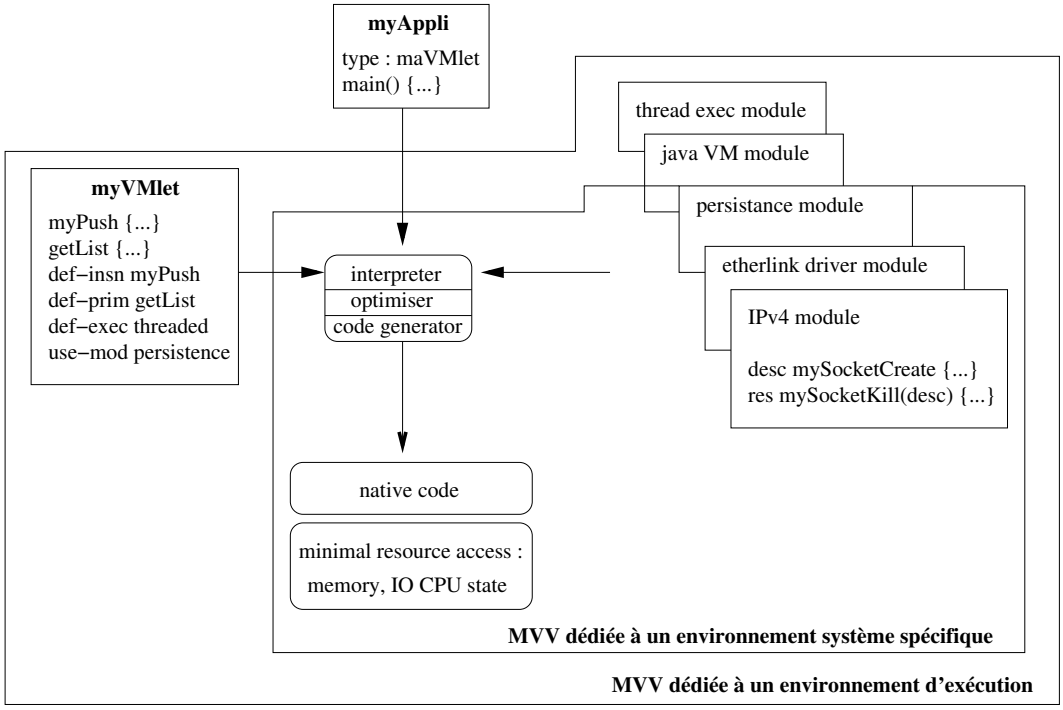

FIG. 2.4 – Architecture de la MVV

variable, un événement est créé au niveau de base puis propagé jusqu'au niveau méta. À ce niveau, ces événements sont traités par des méta-objets.

MetaXa permet donc la création de méta-objets destinés à contrôler l'exécution du niveau de base depuis le niveau méta. Lors de leur instanciation, ces méta-objets sont initialisés avec toutes les informations nécessaires provenant du niveau de base et permettant en particulier le contrôle de l'invocation de méthodes. Plusieurs méta-objets peuvent être attachés à un même objet du niveau de base. De manière similaire, un même méta-objet peut contrôler plusieurs objets du niveau de base. L'attachement peut être réalisé sur des références, des instances de classes ou encore des classes. Dans le cas de l'attachement sur les références, afin d'éviter un enchevêtrement du code de configuration et du code de création des méta-objets, ces derniers suivent la règle suivante : si une référence est écrite dans une variable d'instance, le méta-objet de l'ancienne référence est détaché de celle-ci puis est attaché à la nouvelle référence.

MetaXa permet de chaîner les méta-objets entre eux. L'ordre de leur chaînage a de l'importance. Le code contenu dans ces méta-objets est exécuté avant que le code de l'objet du niveau de base qui est rattaché ne soit exécuté :

l'exécution au niveau de base est ainsi suspendue durant l'exécution des méta-objets.

L'attachement des méta-objets, dans MetaXa, peut aussi être réalisé sur des parties de la machine virtuelle comme le chargeur de classes, les moniteurs, les processus légers ou encore les modules de création d'objets. De tels attachements permettent de redéfinir ou d'ajouter de nouveaux comportements à ces parties de la machine virtuelle. Chacun des méta-objets accède à la machine virtuelle *via* une interface Java qui contient des méthodes pour la réflexivité structurelle. Cette interface permet au programmeur de créer les méta-objets dont il a besoin. Pour cela il créé une classe qui dérive de la classe Java `MetaObjet`. Tous les méta-objets créés possèdent donc un type dynamique qui dérive de ce type commun. L'exemple suivant illustre le lien entre les méta-objets et les objets qu'ils contrôlent. Il affiche une trace des méthodes du niveau de base qui sont exécutées :

```
public class MetaTrace extends MetaObject {
  //méthode d'attachement du méta-objet
  public void attachObject(Object o, String methodNames[]) {
     ...
  }

  //traitement d'un événement pour les méthodes sans retour
  public void eventMethodEnterVoid(EventMethodCall event) {
    System.out.println("Method" + event.methodname + "called!");
    ...
    continueExecutionVoid(event);
  }

  //traitement d'un événement pour les méthodes ayant un retour
  public Object eventMethodEnter(EventMethodCall event) {
    System.out.println("Method" + event.methodname + "called!");
    ...
    Object returnValue = continueExecution(event);
    System.out.println("returned: " + returnValue.toString());
    return returnValue;
  }

  ...
}
```

Pour un objet `obj` et un tableau contenant le nom de ses méthodes, l'attachement se réalise *via* un appel du genre :

```
(new MetaTrace()).attachObject(obj, methods) ;
```

La machine virtuelle MetaXa permet donc un contrôle de l'exécution *via* l'utilisation de méta-objets. Elle permet de modifier la sémantique d'exécution des *méthodes* mais non de ses *composants*. Elle ne peut par exemple modifier la représentation des données de l'application par le changement de son tas.

2.3.2.5 SOUL

SOUL [WD01] est un langage logique proche de PROLOG qui permet d'observer un programme écrit en SmallTalk tandis que lui même est aussi implanté en SmallTalk. SOUL permet d'accéder à la représentation de son interprète et de modifier celui-ci. Les termes manipulés par SOUL sont ainsi :

- des termes logiques,
- des objets SmallTalk, représentant ou non une partie de l'état de son interprète.

Par exemple, dans l'expression SOUL `method([Array],?m)`, l'interprète manipule `Array` (une entité SmallTalk et non SOUL). De même, dans l'expression SOUL `method([SOULVariable],?m)`, l'interprète manipule `?m`, une variable (une entité du langage de base) et `SOULVariable`, une méta-entité de l'implantation SmallTalk de SOUL, c'est-à-dire appartenant à la réification de l'état d'exécution.

L'interprète de SOUL est ainsi réflexif. Mais il est aussi dit *symbiotique* car en plus d'être capable de manipuler la réification de son état d'exécution, il est aussi capable de manipuler des objets du langage dans lequel il est écrit, c'est-à-dire le méta-langage. Les possibilités de réflexivité du langage SOUL sont alors multiples :

- l'introspection : les termes logiques de SOUL peuvent raisonner sur d'autres termes SOUL.
- l'introspection symbiotique : SOUL peut raisonner sur des termes SmallTalk.
- l'intercession symbiotique : SOUL peut modifier sa propre implantation en changeant des objets SmallTalk réifiants son interprète SmallTalk.

Pour procéder à une évaluation uniforme des termes des expressions qu'il manipule, le schéma suivant est utilisé, T représentant l'ensemble des termes Soul et O l'ensemble des objets Smalltalk :

. $Up : O \rightarrow T$ (*transformation d'un objet Smalltalk en un terme Soul*)
  - si x est un terme alors $Up(Down(x)) = x$,
  - sinon (x est un objet SmallTalk) Up transforme cet objet SmallTalk en un terme logique

. $Down : T \rightarrow O$ (*transformation d'un terme Soul en un objet Smalltalk*)
  – si x est un terme alors $Down(x)$ retourne l'implantation SmallTalk de ce terme,
  – sinon (x est un objet SmallTalk) alors $Down(Up(x)) = x$

Ainsi, pour une évaluer une expression, SOUL procède à la fonction *Down* sur tous les termes de cette expression, puis applique la fonction *Up* au résultat de cette évaluation. Bien que l'adaptation de programme ne soit pas au cœur de ces travaux, le mécanisme *Up/Down* permettant l'introspection et l'intercession symbiotique sera un outil précieux pour Corosol.

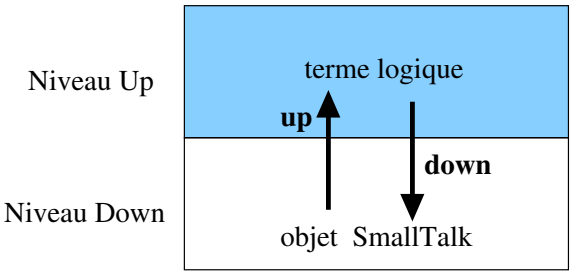

FIG. 2.5 – Schéma Up/Down

2.3.2.6 Agora

Agora[Meu98] est un langage à prototypes réflexif dans lequel structures de données et programmes sont des objets. Il est de construction minimal car l'envoi de messages est son unique structure de contrôle. Cependant, son protocole à méta-objets permet d'étendre le langage de façon incrémentale. Agora peut donc définir ainsi une famille de langages. De plus, comme dans tout langage à prototypes, les extensions apportées ne sont effectuées que par des modifications sur des *objets*. Ainsi, les propriétés d'héritage, de clonage et de réflexivité, peuvent être ajoutées séparément *via* de simples envois de messages particuliers, appelés *messages réificateurs*[1]. En définissant de nouvelles implantations de messages réificateurs, Agora peut ainsi être étendu et définir de nouveaux langages.

Par exemple, durant l'exécution, l'envoi des messages réificateurs `VIEW` et `MIXIN` ajoutent respectivement deux types d'héritage, les *vues* et les *mixins*. Les *vues* ajoutent à un objet de nouvelles méthodes ou de nouveaux champs, sans en modifier sa structure interne. Un nouvel objet est simplement créé

[1] *reifier messages*

et l'objet receveur du message `VIEW` devient son parent. Les *mixins* [SP] permettent de réaliser les mêmes opérations d'héritage que les vues, mais cette fois l'objet receveur du message `MIXIN` est modifié. Les nouvelles méthodes ou les nouveaux champs sont ajoutés physiquement à l'objet. De même, le clonage des objets de l'application s'effectue également par envoi du message réificateur `CLONING`. Contrairement à Java, Agora ne prévoit pas d'opérateur natif pour effectuer cette opération. En Java, il s'agit de la méthode native *clone* de la classe `java.lang.Object` (bien qu'elle puisse être redéfinie par des sous-classes, elle est dans un premier temps dépendante de l'implantation de la machine virtuelle sous-jacente).

L'interprète de Agora est implanté *via* un langage orienté-objets. Parmi les objets composants son architecture, on distingue en particulier les *objets Agora*, qui *implantent* les objets de l'application. Tous les objets du niveau méta implantent la méthode `up` qui retourne un objet Agora. Celui-ci représentant un objet du niveau de base, il est visible depuis l'application. Les objets Agora possèdent deux méthodes supplémentaires :

- `send`, qui implante l'envoi de messages,
- `down`, qui transforme un objet de l'application en un objet de l'interprète ; si l'objet Agora est déjà un objet de l'interprète, l'opération est idempotente et l'objet retourné par `down` est cet objet ; dans le cas contraire, l'objet retourné est l'objet de l'interprète qui implante l'objet du niveau de base.

L'envoi de message à un objet `obj` du niveau de base provoque l'évaluation de l'expression associée au niveau méta : la méthode `send` est appelée sur l'objet Agora représentant `obj`. Tous les objets Agora qui représentent cette expression sont ensuite transformés en objets de l'interprète *via* la méthode `down`. Le résultat de cet appel est un nouvel objet de l'interprète. Il est ensuite transformé en un objet Agora *via* sa méthode `up`. Il s'agit alors du résultat de l'envoi du message au niveau de base.

Ainsi, Agora possède un mécanisme *up / down* similaire à SOUL [WD01]. En effet, Agora permet d'accéder à la représentation de son interprète durant l'exécution et également de modifier les composants de son architecture par des objets de Agora. Dans un premier temps, ce mécanisme est implicite lorsqu'il s'agit d'utiliser les valeurs numériques ou les messages réificateurs, car ces éléments sont des parties de l'interprète du langage. Mais il peut être rendu explicite au niveau du langage, *via* l'utilisation de deux messages : `UP` et `DOWN`. Le message `UP` est utilisé pour réifier les objets de l'interprète et les représenter sous la forme d'objets de l'application. Le message `DOWN` est utilisé pour l'opération inverse, c'est-à-dire transformer un objet de l'application en un objet de l'interprète. D'autres messages sont également utilisés pour transformer une expression du langage en un objet Agora (c'est-à-dire la

transformation d'un arbre d'expression utilisé par l'interprète en un objet Agora). L'opération inverse est également possible. Il s'agit dans ce cas de transformer un objet Agora en une expression du langage, *via* l'envoi d'un message.

2.3.2.7 Guaraná

Guaraná [OB99, OB98, OGB98] est une extension de *Kaffe Open VM*, une implantation libre (au sens *open source*) de la machine virtuelle Java. Dans son implantation, Guaraná modifie l'interprète de base de *Kaffe Open VM* afin d'y introduire différents mécanismes d'interception. Ces mécanismes permettent le contrôle de l'exécution *via* la création de méta-objets. L'objectif du projet Guaraná est ainsi de faciliter la gestion de ces méta-objets et en particulier leurs compositions. Le protocole à méta-objets de Guaraná est écrit dans le langage Java.

Guaraná limite l'association entre les méta-objets en utilisant le modèle de conception de composite (*Composite Pattern*). Ainsi, à un moment donné, un objet du niveau de base n'est associé qu'à un unique méta-objet et ce dernier peut être en particulier un *composeur*. Un composeur assure la gestion de plusieurs méta-objets qui peuvent être aussi des composeurs.

Chaque composeur délègue les opérations du niveau de base entre plusieurs méta-objets. Outre leurs effets de bords, chaque méta-objet peut calculer un résultat. Après avoir délégué des requêtes du niveau de base aux différents méta-objets, un composeur est chargé de regrouper les différentes réponses, qu'il peut composer pour calculer une valeur qui sera transmise au niveau de base.

Ainsi, le rôle d'un composeur est d'assurer la gestion des requêtes/réponses entre les différents méta-objets, mais aussi avec le niveau de base.

L'avantage de l'utilisation des composeurs est de pouvoir aider dans la résolution des conflits entre méta-objets incompatibles entre-eux. La résolution de tels conflits est gérée en découpant les mécanismes du niveau méta en quatre étapes. Dans la première, si aucun méta-objet n'est associé à un objet du niveau de base, alors l'exécution continue au niveau de base sans passer par le niveau méta. Cela n'entraîne donc aucun surcoût pour l'exécution. Dans la seconde étape, si un méta-objet est sollicité par le niveau de base, il produit alors un objet qui est le résultat de l'opération qu'il vient d'effectuer. Cet objet est la réification du résultat attendu par le niveau base. Ce résultat est ensuite dé-réifier avant d'être retourné au niveau de base. La troisième étape considère que la réponse d'un méta-objet à une requête du niveau de base peut ne pas avoir de résultat. Dans ce cas, ce méta-objet répond en demandant l'exécution d'une opération du niveau de base, et en

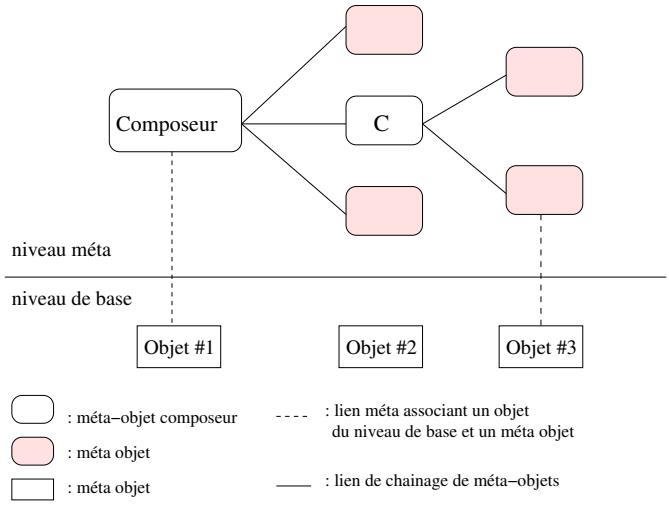

FIG. 2.6 – Les méta-objets dans Guaraná

particulier celle qu'il devait contrôler. A cette réponse, il ajoute une notification précisant au niveau de base s'il est intéressé au non par une éventuelle gestion du résultat de cette opération. Finalement, l'exécution de celle-ci est effectuée dans la quatrième étape. Son résultat est éventuellement délivré au méta-objet, si ce dernier en a demandé la gestion. Dans ce cas, il transmet au niveau de base la modification de ce résultat.

Un tel découpage permet de séparer la gestion et la configuration des méta-objets, et donc de l'organisation du niveau méta, de leurs implantations. Guaraná permet ainsi une meilleure réutilisation de ceux-ci.

La gestion des méta-objets par cet outil permet un paramétrage et un contrôle de l'exécution. Ainsi, cette dernière peut être adaptée en fonction des besoins de l'application.

2.4 Conclusion et objectifs

Les machines virtuelles qui permettent à un programme de s'adapter en fonction des nouveaux besoins se classent donc en deux catégories :

- celles qui permettent une adaptation avant démarrage, *via* le choix (par reprogrammation ou non) de certains composants,
- celles qui permettent une adaptation durant l'exécution, *via* l'utilisation de méta-objets chargés d'un contrôle fin de l'exécution.

La première catégorie présente l'avantage d'être des machines virtuelles fortement modulaires et «prêtes à l'emploi» dès le démarrage. Cependant, l'adaptation dynamique est exclue, ce qui est permis par la seconde catégorie. Mais celles-ci, dans beaucoup de cas, sont des machines virtuelles *ad-hoc* et non portables : l'application et le code de son adaptation ne peuvent être utilisés sur une autre machine virtuelle.

Notons cependant le cas particulier de 3-Lisp. Bien que celui-ci soit un interprète *ad-hoc* d'un programme Lisp, il définit des idées importantes que nous utiliserons. Cet interprète peut être enrichi et modifié de façon dynamique. Ainsi, dans celui-ci, l'adaptation dynamique de programme est effectuée *via* réflexivité comportementale. L'interprète est modifié parce qu'il est lui-même interprété par un autre interprète et ainsi de suite. 3-Lisp définit une idée fondamentale : le code réflexif qui modifie la sémantique de l'exécution du programme se déroule au même niveau que celui-de l'interprète de ce programme. Ces notions sont aussi à l'origine de notre propre machine virtuelle.

De même, nous avons abordé le cas de la machine virtuelle Smalltalk. Ce langage est dans sa définition entièrement réflexif : les éléments qui le composent, les éléments décrivant sa sémantique et ceux de son exécution sont réifiés. Par exemple, nous avons décrit l'interprète SmallTalk du langage SOUL qui est réflexif et symbiotique. Même si adapter le programme exécuté et/ou sa plate-forme d'exécution n'est pas le centre de ces travaux, les idées développées autour de ce langage sont importantes. En particulier, la méthode d'évaluation des expressions procède à une analyse uniforme de chacun des termes *via* le mécanisme *Up/Down*. Ainsi, peu importe la provenance des paramètres analysés (soit du niveau de base, élément du langage, ou du niveau méta, élément de l'interprète), chacun d'entre eux est analysé de la même manière par l'interprète. Cette notion est aussi utilisée dans nos travaux pour l'évaluation des méthodes en fonction de la provenance de l'objet cible.

Nous avons développé notre machine virtuelle avec ce double objectif :

- d'une part obtenir une architecture suffisamment modulaire pour que le choix des composants internes puissent être décidé avant démarrage, ce qui permettra l'adaptation «statique» d'une application,
- d'autre part avoir une machine virtuelle portable permettant l'adaptation dynamique des programmes.

Nous désirons donc de réaliser une plate-forme d'exécution dynamiquement configurable depuis l'application pour que celle-ci dernière puisse s'adapter en fonction de ses besoins en configurant la machine virtuelle. On utilisera pour cela la réflexivité, comme dans 3-Lisp. En cela, notre objectif possède des points communs avec le projet VVM [FPR98, PFSB00, Fol00, FBPS], le projet 3-Lisp [Smi82, Smi84], mais également avec des implantations du langage Smalltalk telle que Squeak [IKM+97]. De plus, afin d'assurer la portabilité et de permettre à l'application de décrire les modifications à apporter à la machine virtuelle dans son propre langage, nous proposons une architecture entièrement Java d'une machine virtuelle Java : *Corosol*.

Corosol étant un programme Java, il est exécuté par une machine virtuelle Java standard, ce qui garanti la portabilité. Dans Corosol, chaque composant de la machine virtuelle est représenté par un composant Java. L'application que Corosol exécute peut redéfinir ces composants en utilisant ses propres objets *via* une interface d'introspection.

Ces idées seront examinées à partir du chapitre 6 de cette thèse. Nous y détaillerons quelques propriétés de cette architecture et comment elle permet l'adaptation de programmes. Présentons tout d'abord Java, sa machine virtuelle ainsi que son code compilé, le *bytecode* Java. Cela fera l'objet des chapitres 3 à 4.

Deuxième partie

Java et sa machine virtuelle

Chapitre 3

Les concepts du langage Java

3.1 Introduction

Avant d'aborder les machines virtuelles qui permettent à une application de s'adapter en fonction de nouveaux besoins, nous présentons, dans ce chapitre et dans le suivant, le langage Java et sa machine virtuelle. Nous en rappelons les caractéristiques les plus importantes. Nous commencerons par un bref historique du langage.

3.2 Historique

Le langage Java a vu la jour dans les années 1990 avec James Gosling, au sein de la firme Sun Microsystems. Il souhaitait développer un langage de programmation pouvant permettre de programmer des appareils variés comme des téléviseurs, des magnétoscopes, des téléphones ou des systèmes embarqués. Ce langage devait aussi permettre de les contrôler et de les rendre interactifs. C'est ainsi que le langage Oak fut créé. Au vu des objectifs fixés, Oak possédait une caractéristique vraiment intéressante : il pouvait s'exécuter quelle que soit la plate-forme matérielle. Mais ce projet fut un échec.

Par la suite Bill Joy (co-fondateur de la firme Sun Microsystems) proposa une nouvelle version de Oak appelée Java (en rapport avec l'île de Java d'où les programmeurs puisaient le café nécessaire à leur création). Son objectif était de fournir un langage de programmation pouvant s'exécuter indépendamment de la plate-forme, dans le contexte des machines et des logiciels hétérogènes qui composent l'Internet.

Ainsi, en 1994, Sun développa le navigateur Web HotJava. L'interprète de code Java (appelé *machine virtuelle*) qu'il intégrait était capable de faire fonctionner des *applets*, applications écrites en Java et intégrées aux pages

Web, quel que soit le système d'exploitation sous-jacent. Mais, en 1995, c'est Netscape qui fut l'un des éléments essentiels à la popularisation de Java en intégrant son propre interprète de programme Java dans son naviguateur Web.

Après de très nombreuses modifications visant à améliorer le système (voir la figure 3.1), Java est devenu plus qu'une solution Internet. Il est désormais un langage utilisé pour toutes sortes de développement.

Java 1.0 Première version stable de Java (1995), elle fut surtout orientée programmation Web et adaptée à tous les navigateurs.

Java 1.1 Deux ans plus tard, la version 1.1 est disponible. Nettement améliorée, cette version implante des nouveaux concepts tels que les JavaBeans (composants d'applications graphique), les fichiers JAR (archives de fichiers Java) ou JDBC pour l'accès aux bases de données.

Java 2 Standard Edition 1.2 Un an plus tard, cette version implante de nombreuses nouveautés telles que JDBC 2.0, les collections (des conteneurs d'objets comme les listes, les ensembles ou encore les tables de hachage), Swing (une nouvelle API pour la création d'interfaces graphiques), ou Java2D. À partir de cette version, on parlera de Java 2 version x.x.

Java 2 Standard Edition 1.4 Cette version a encore évolué et implante de nouvelles fonctionnalités telles que le support XML, JDBC 3.0 ou encore le paquetage *java.nio* (une nouvelle API pour la gestion des entrées/sorties).

Java 2 Standard Edition 5.0 «Tiger» Cette dernière version apporte quelques modifications sur la syntaxe du langage, bien que celles-ci ne soient pas obligatoires (autoboxing, annotation, collections typées, classes paramétrables à l'image des templates de C++,...). Les performances de l'interprète sont améliorées et de nouveaux paquetages apparaissent. C'est la version actuelle.

FIG. 3.1 – Historique des versions de Java

3.3 Un langage portable

Java doit beaucoup de sa popularité à sa portabilité. Par portabilité, il faut comprendre la possibilité pour un programme Java de s'exécuter indépendamment du système d'exploitation sous-jacent. Les sections suivantes détaillent cette propriété importante et examinent ses deux fondements qui y contribuent, savoir :

- le code compilé, appelé *bytecode*, ainsi que
- la *machine virtuelle* Java.

3.3.1 Un code compilé portable

Le résultat de la compilation d'un programme Java est un code intermédiaire appelé *bytecode*. Ce dernier est indépendant de la plate-forme matérielle et logicielle (que l'on soit sur un Pentium, un PowerPC, un Sparc ou sur un Alpha, sous Windows, MacOS, Solaris ou Linux, etc...). Par exemple, dans un navigateur Web, sur un Palm Pilot, voire sur une carte à puce (avec cependant un prétraitement et quelques restrictions, vu la limitation des ressources de ces cartes), l'exécution s'effectue avec le même code. Cette indépendance garantit la portabilité des applications écrites en Java.

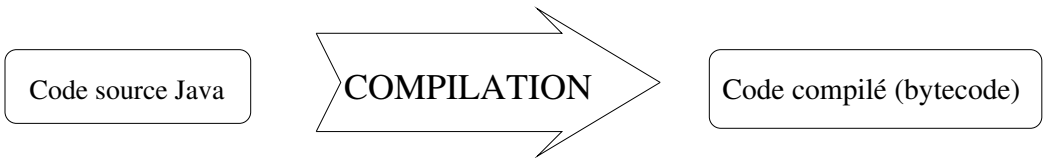

FIG. 3.2 – La compilation d'un programme écrit en Java

Considérons la portion de code suivante écrite en Java. Elle effectue l'affichage à l'écran de la chaîne de caractères `Hello world!` :

```
public class HelloWorld {
  public static void main(String[] args) {
    System.out.println("Hello World!");
  }
}
```

La méthode `main` est ainsi compilée en un code intermédiaire comprenant ente autres une séquence d'instructions qui ressemble à ce qui suit :

```
getstatic 2
ldc 3
```

```
invokevirtual 4
return
```

Nous reviendrons plus en détails sur les instructions de *bytecode* dans les chapitres suivants. Celles-ci sont exécutées par la machine virtuelle Java.

3.3.2 Une exécution portable

En général, le *bytecode* n'est pas directement exécuté par un processeur physique. Aussi, une couche logicielle est introduite entre ce *bytecode* et la machine hôte sur laquelle l'exécution doit se dérouler. Celle couche logicielle est appelée la *machine virtuelle Java*.

Sa principale fonction est d'exécuter les séquences d'instructions de *bytecode*. La sémantique de chaque instruction de bytecode est décrite dans la spécification de la machine virtuelle Java fournie par Sun [LY99].

Un programme Java n'est compilé qu'une fois pour toutes mais son *bytecode* peut donc être exécuté sur n'importe quel système, pourvu que ce dernier possède sa propre implantation de machine virtuelle Java. La figure 3.3 illustre cette idée.

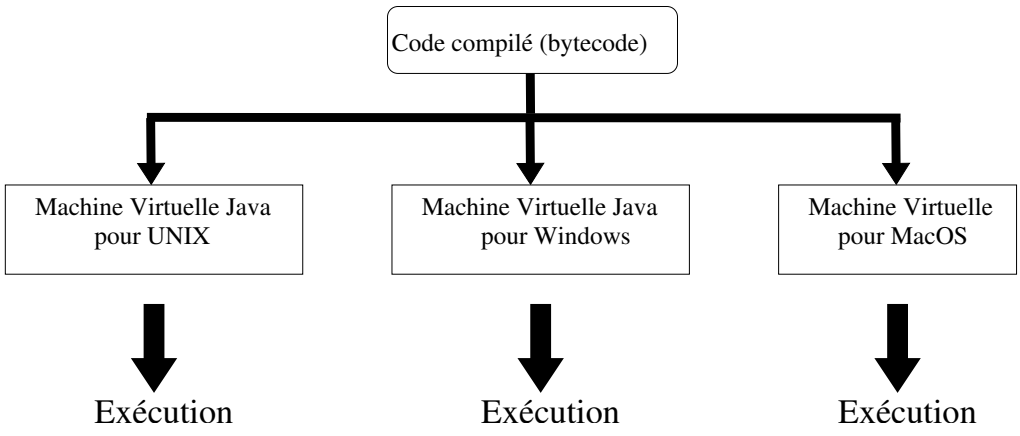

FIG. 3.3 – Exécution *via* une machine virtuelle

L'exécution d'un programme Java est donc portable en raison du caractère interprété de celui-ci, mais aussi en raison de son code compilé. Nous examinons de plus près ce dernier dans la prochaine section.

3.4 Le code compilé (*bytecode*)

Cette section aborde le code compilé d'un programme Java : le *bytecode*. Nous aborderons sa structure en tant que fichier, que nous désignerons par

fichier `class`. Notre objectif n'est pas de détailler en profondeur ce code compilé, mais plutôt de montrer où et comment des changements simples peuvent y être appliqués pour la génération de *bytecode* qui nous sera utile au sein de l'architecture de notre machine virtuelle Java, Corosol.

3.4.1 Structure générale d'un fichier `class`

Dans un système de fichiers, le code compilé d'un programme Java est contenu sous la forme d'un fichier que nous désignons par fichier `class`. Il est chargé et vérifié par la machine virtuelle avant l'exécution du code qu'il contient (le chapitre suivant expliquera cela plus en détails). Un tel fichier est composé de plusieurs parties logiques données par la figure 3.4 (u2 et u4 représentant respectivement deux et quatre octets).

```
ClassFile {
        u4 magic;
        u2 minor_version;
        u2 major_version;
        u2 constant_pool_count;
        cp_info constant_pool[constant_pool_count-1];
        u2 access_flags;
        u2 this_class;
        u2 super_class;
        u2 interfaces_count;
        u2 interfaces[interfaces_count];
        u2 fields_count;
        field_info fields[fields_count];
        u2 methods_count;
        method_info methods[methods_count];
        u2 attributes_count;
        attribute_info attributes[attributes_count];
    }
```

FIG. 3.4 – Structure générale des fichiers au format `class`

La table `constant_pool` est une des parties qui contient le plus d'informations. Nous la décrivons dans la section suivante.

3.4.2 La table `constant_pool`

La table `constant_pool` est utilisée par la machine virtuelle lorsque celle-ci a besoin d'obtenir la valeur d'une constante numérique ou une chaîne de caractères représentant une référence symbolique. Tout le *bytecode* fait référence à des entrées de cette table. Ainsi, en particulier, certaines instructions de la machine virtuelle Java ont pour opérandes des indices d'entrées cette table ; les instructions réalisant les appels de méthodes et les accès aux champs, par exemple. De tels indices pointent vers des entrées contenant les chaînes de caractères représentant le nom, la signature et le nom de la classe d'une méthode ou d'un champ.

Comme nous l'examinerons plus loin, la machine virtuelle dispose d'un mécanisme permettant la transformation de telles références symboliques en références directes sur des données concrètes. Ce mécanisme appelé *résolution dynamique* sera examiné au chapitre suivant.

3.4.2.1 Description

Reconsidérons l'exemple `HelloWorld.java` du chapitre précédent :

```
class HelloWorld {
  public static void main(String[] args) {
    System.out.println("Hello world!");
  }
}
```

Ce fichier est compilé au format `class` en `HelloWorld.class`. La première instruction qu'il contient est `getstatic`. Elle a pour objectif l'accès à la valeur d'une variable de classe. Son opérande est l'indice d'une entrée de la table `constant_pool` dont le type est `CONSTANT_FIELD_info` qui décrit :

- le nom du champ auquel l'instruction `getstatic` accède (`out`),
- le descripteur de ce champ, qui est une chaîne de caractères décrivant le type de celui-ci (`java.io.PrintStream` dans notre exemple) et,
- le nom de la classe où le champ est défini (la classe `java.lang.System` dans notre exemple).

Ensuite, la seconde instruction est `ldc`. Son objectif est de manipuler les chaînes de caractères (dans notre exemple, il s'agit de `"HelloWorld"`). Elle possède un opérande qui est un indice de la table `constant_pool` qui désigne une entrée de type `CONSTANT_String_info` représentant un objet java de la classe `java.lang.String`. Finalement, la dernière instruction `invokevirtual`, utilisée pour les appels de méthodes, se compose également d'un opérande dont la valeur est un indice vers une entrée de type

`CONSTANT_Methodref_info`. Elle décrit le nom, la signature et la classe de la méthode à appeler (dans notre exemple, il s'agit de la méthode `println` de la classe `java.lang.System`).

L'ensemble des différents types d'entrées de la table `constant_pool` est décrit au tableau 3.1.

Type d'entrée	**Référence symbolique désignée**
`CONSTANT_Class_info`	Interface, classe ou tableau.
`CONSTANT_Methodref_info`	Méthode d'instance ou de classe.
`CONSTANT_InterfaceMethodref_info`	Méthode d'interface.
`CONSTANT_Integer_info`	Constante numérique de type `int`
`CONSTANT_Float_info`	Constante numérique de type `float`
`CONSTANT_Long_info`	Constante numérique de type `long`
`CONSTANT_Double_info`	Constante numérique de type `double`
`CONSTANT_String_info`	Instance de la classe `java.lang.String`.
`CONSTANT_NameAndType_info`	Partagée par les entrées de type `CONSTANT_XXXref_info`.
`CONSTANT_Utf8_info`	Chaîne de caractères encodée au format UTF-8.

TAB. 3.1 – Les entrées de la table `constant_pool`.

À l'exception de l'entête (`magic`), du numéro de version (`minor_version` et `major_version`) et du modificateur d'accès de la classe représentée, c'est-à-dire son niveau de visibilité (`access_flag`), chacune des parties d'un fichier au format `class` fait référence à la table `constant_pool`. Par exemple, la liste des interfaces implantées par la classe représentée par ce fichier est désignée par un ensemble d'entiers correspondants aux indices d'entrées de type `CONSTANT_Class_info`. Il en est de même pour la superclasse, qui est définie dans fichier `class` par un indice vers un même type d'entrée (*cf.* le champ `superclass` de la figure classfile.

Les champs et les méthodes sont représentés dans le fichier `class` respectivement par les parties `field_info` et `method_info` (voir figure 3.4). Le nom, le type et la signature y sont aussi décrits en termes d'indices de la table `constant_pool`. Ces parties contiennent aussi d'autres informations importantes. C'est le cas de `method_info` qui contient les instructions de chaque méthode. Ces informations sont appelées *attributs*. Nous les décrivons dans la section suivante.

3.4.3 Les attributs (*attributes*)

Les attributs sont des informations supplémentaires sur certaines parties d'un fichier au format `class`. Ils correspondent à chacune des structures `attribute_info` (voir figure 3.4), composé d'un entier déterminant la taille des données de cet attribut et d'un indice d'une entrée `CONSTANT_Utf8` dans la table `constant_pool` qui désigne son type.

Parmi les attributs les plus importants, on distingue tout d'abord celui de type `ConstantValue`. Il est contenu dans les structures `field_info` qui décrivent les variables de classe (notée `static` dans le code source Java). Il est représenté par une suite d'octets désignant un indice d'entrée dans la table `constant_pool`. Cette entrée contient la valeur avec laquelle le champ doit être initialisé lorsque la classe dans laquelle il est déclaré est chargée par la machine virtuelle. Le type d'une telle entrée sera par exemple `CONSTANT_long_info` pour une variable de classe de type `long` et celui d'une variable de type `java.lang.String` sera `CONSTANT_String_info`.

L'attribut de type `Code` est le plus important parmi ceux contenus au sein de chacune des structures `method_info`. Il contient les instructions, la table des exceptions et la taille de la pile d'exécution d'une méthode.

Selon la spécification de la machine virtuelle Java [LY99], il est possible de créer autant d'attributs que l'on désire, en plus de ceux qu'elle impose. Certains attributs sont ainsi *optionnels*. Par exemple, l'attribut `LineNumberTable` est utilisé par les débogueurs pour faire l'association entre une ligne du code source et un endroit dans le flot d'instructions. L'attribut `Deprecated`, quant à lui, s'il est présent dans la liste des attributs des structures `ClassFile`, `method_info` ou `field_info`, désigne respectivement une classe, une méthode ou un champ noté obsolète et donc non utilisable par le programmeur. Un compilateur générera tel ou tel attribut optionnel au sein d'un fichier `class`. Cependant, ces derniers peuvent être ignorés par une machine virtuelle Java, s'ils lui sont inconnus, lorsque celle-ci extrait du fichier `class` les informations d'exécution.

Le tableau 3.2 présente les autres attributs non optionnels.

3.5 Génération de *bytecode* par modifications

Il existe de nombreux outils qui manipulent le *bytecode*, certains pouvant même le générer à la volée durant l'exécution. Parmi eux, on distingue par exemple BIT [LG97], Javassist [Chi00, Chi98], BCEL [Dah01] ou encore ASM [BLC02].

Nous nous intéressons au cas où l'on créé du *bytecode* à partir de celui

Type d'attribut	Description
`Code`	Contient les instructions de *bytecode* et les informations relatives à l'exécution d'une méthode.
`ConstantValue`	Désigne l'entrée de la table `constant_pool` contenant la valeur d'une variable de classe.
`Exceptions`	Décrit les exceptions pouvant être levées par une méthode.
`InnerClasses`	Décrit le nom, le type et le modificateur d'accès d'une classe interne.
`Synthetic`	Désigne n'importe quel attribut généré automatiquement par le compilateur Java.

TAB. 3.2 – Les différents types d'attributs.

d'une classe et *en lui apportant des modifications*. Pour que ces changements soient effectifs, ils doivent être opérés avant son chargement dans la machine virtuelle Java (il est à noter que depuis la version 5 de Java, qui permet un rechargement du *bytecode*, ceci n'est plus vrai). Dans notre machine virtuelle, Corosol, se sera important pour l'exécution (*cf.* chapitre 8).

D'une manière générale, modifier le *bytecode* d'une classe signifie :
- lui ajouter (mais aussi supprimer, renommer) une méthode ou un champ,
- modifier la séquence d'instructions d'une méthode,
- modifier la hiérachie de cette classe (sa super-classe et/ou ses super-interfaces).

Examinons comment réaliser de telles opérations. Encore une fois, la table `constant_pool` sera un élément central dans celles-ci.

3.5.1 Ajouter un champ ou une méthode

Considérons à nouveau la figure 3.4. L'ajout de méthodes est réalisé en créant de nouvelles entrées dans la table `constant_pool`. Ce sont des entrées de type `CONSTANT_Utf8` qui désignent le nom et la signature de ces nouvelles méthodes.

De nouvelles entrées dans le tableau `method_info` sont également ajoutées. Celles-ci contiennent les indices des entrées de la table `constant_pool` précédemment créés et qui décrivent le nom et la signature des nouvelles méthodes. Chacune des nouvelles entrées du tableau `method_info` inclue, et c'est le plus important, la séquence d'octets correspondant à de nouveaux attributs de type `Code`. Ceux-ci contiennent effectivement la liste d'instructions des nouvelles méthodes, mais aussi les informations relatives à l'exécution

comme la taille maximale de la pile ou encore le nombre de *variables locales* utilisées (cette dernière notion est examinée au chapitre suivant).

La même démarche prévaut pour l'ajout d'un champ. Dans ce cas, c'est la table `field_info` qui se voit augmentée d'une nouvelle entrée. Cette dernière peut contenir un éventuel attribut `ConstantValue`, si le champ ajouté est une variable de classe. Cet attribut fait alors référence à une entrée de la table `constant_pool` qui contient la valeur d'initialisation de ce champ.

3.5.2 Modifier la séquence d'instructions d'une méthode

Pour modifier la séquence d'instructions d'une méthode il faut d'abord accéder aux informations contenues dans l'entrée du tableau `method_info` qui décrit cette méthode et en particulier à son attribut `Code`. C'est ce dernier qu'il faut modifier pour changer les instructions de *bytecode*.

En effet, un attribut code se constitue d'une séquence d'octets correspondant aux instructions de la machine virtuelle Java. Modifier les octets de ce tableau signifie modifier les instructions d'une méthode. Cependant, un attribut de type `Code` contient aussi deux informations essentielles pour l'exécution :

- la taille maximale de la pile d'exécution associée à la méthode (cette pile est désignée par le terme *frame* au chapitre suivant) et,
- le nombre maximal de variables locales de cette méthode.

Après avoir modifié les instructions d'une méthode, il faut donc veiller au recalcul de ces deux nombres.

3.5.3 Modifier la hiérarchie d'une classe

Si l'on désire modifier la hiérarchie d'une classe, il faut aussi modifier la table `constant_pool`.

Dans un premier temps, de nouvelles entrées dans la table `constant_pool` sont créés. Chacune est du type `CONSTANT_Class_info` et est créée à moins qu'elle n'existe déjà dans cette table. Chacune correspond à la description d'un nouveau type que nous désirons voir intervenir dans la hiérarchie de la classe.

Dans un second temps, on procède à une renumérotation d'indices. Considérons à nouveau la figure 3.4. Pour modifier la superclasse, il faut changer la valeur de l'indice `super_class`. Celui-ci sera un index vers une entrée de la table `constant_pool` de type `CONSTANT_Class_info` précédemment ajoutée. De même, la liste des super-interfaces directes d'une classe peut être modifiée en changeant la valeur des indices contenus dans la table `interfaces`, cha-

cun d'entre eux pointant vers une entrée `CONSTANT_Class_info` de la table `constant_pool`.

3.6 Conclusion

Dans ce chapitre, nous avons rappelé une propriété importante du langage Java : sa portabilité. Nous avons examiné succintement deux éléments de Java qui le permettent : son code compilé, appelé *bytecode* et sa machine virtuelle.

Dans ce chapitre, nous avons abordé la structure du *bytecode* d'une classe. Celui-ci est sous la forme d'un fichier `class` très structuré. Cette mise en forme du *bytecode* permet de modifier celui-ci de façon simple, avant chargement au sein de la machine virtuelle. Nous avons donc examiné les bases de telles modifications du code compilé. Cela sera important pour notre propre machine virtuelle Java, Corosol (*cf.* chapitre 8). Mais avant, décrivons l'architecture d'une machine virtuelle Java selon la spécification de Sun Microsystem [LY99].

Chapitre 4

La machine virtuelle Java

4.1 Introduction

Dans ce chapitre, nous présentons la machine virtuelle Java décrite par la spécification de Sun Microsystems [LY99]. Nous détaillons son architecture et comment s'exécute un programme Java. Cela nous aidera à comprendre l'architecture et le fonctionnement de notre machine virtuelle Java, Corosol.

4.2 Présentation

La machine virtuelle Java est chargée d'exécuter un programme Java. Le chapitre précédent a montré que celui-ci est chargé au sein de cette machine sous la forme de structures au format `class`, aussi désignées par *bytecode*.

La machine virtuelle Java est décrite par la spécification abstraite de Sun Microsystems dans le livre *The Java Virtual Machine Specification* [LY99]. Elle est une plate-forme d'exécution portable. En effet, le *bytecode* d'un programme Java peut s'exécuter sur n'importe quelle implantation de cette spécification abstraite. D'ailleurs, de nombreuses implantations de celle-ci ont été écrites. Par exemple, il en existe pour différents systèmes d'exploitation. Il existe aussi des implantations pour des plate-formes particulières, par exemple les cartes à puces [Jav]. La machine virtuelle Java est aussi un environnement d'exécution sûr, car elle gère elle-même celui-ci. Ce sont deux avantages très importants dans un cadre composé de machines et d'applications hétérogènes comme l'Internet.

Cependant, des inconvénients sont à noter. Le premier est un surcoût de l'exécution, conséquence de l'interprétation du code de l'application. Sans compilation à la volée de l'exécution au moyen d'un compilateur *juste à temp* (de l'anglais *Just In Time* ou simplement JIT), l'exécution est plus lente par

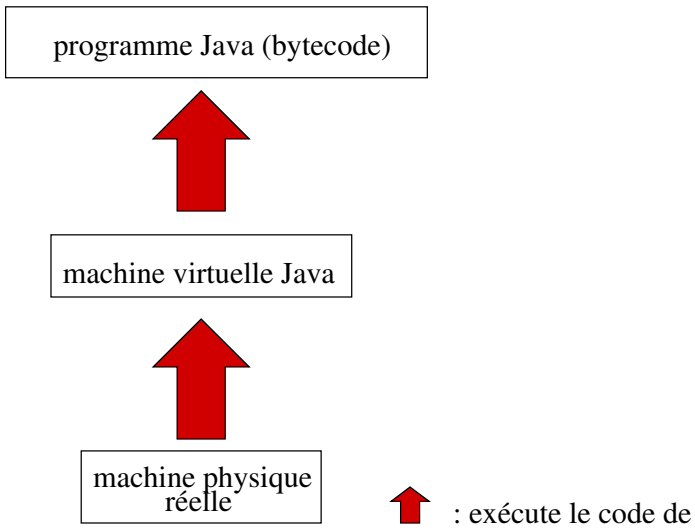

FIG. 4.1 – Exécution d'un programme Java

rapport à du code natif (de plusieurs dizaines voire centaines de fois) et consomme beaucoup plus de ressources systèmes (que ce soit en temps processeur ou en occupation mémoire). Le second touche à la mise en œuvre d'une machine virtuelle. Par exemple, plusieurs dizaines de milliers de lignes de langage C sont nécessaires pour une implantation standard. Quant au dernier inconvénient, il s'agit du manque de configurabilité. Les machines virtuelles existantes ne permettent pas à une application de s'adapter à sa plate-forme d'exécution (l'inverse étant aussi faux). Par exemple, les implantations qui contiennent des spécificités hors-norme sont souvent *ad-hoc* pour satisfaire à un cadre donné, comme la persistance [LMG00], le parallélisme de l'exécution [DA01a] ou encore la réflexivité comportementale [GK98].

Avantages :	Inconvénients :
- portabilité de l'exécution - sûreté de l'exécution	- lenteur de l'exécution (sans compiltateur JIT) - mise en œuvre difficile - peu de configurabilité

FIG. 4.2 – Avantages et inconvénients de l'utilisation d'une machine virtuelle

4.3 Architecture

La spécification de Sun Microsystems [LY99] définit ce qui est requis pour toute implantation de la machine virtuelle Java. Elle décrit de façon stricte le comportement externe d'une instance de la machine virtuelle. Elle en présente l'architecture en termes de sous-systèmes, de zones de mémoire, de types de données et aussi d'instructions. La figure 4.3 donne une vue d'ensemble de celle-ci.

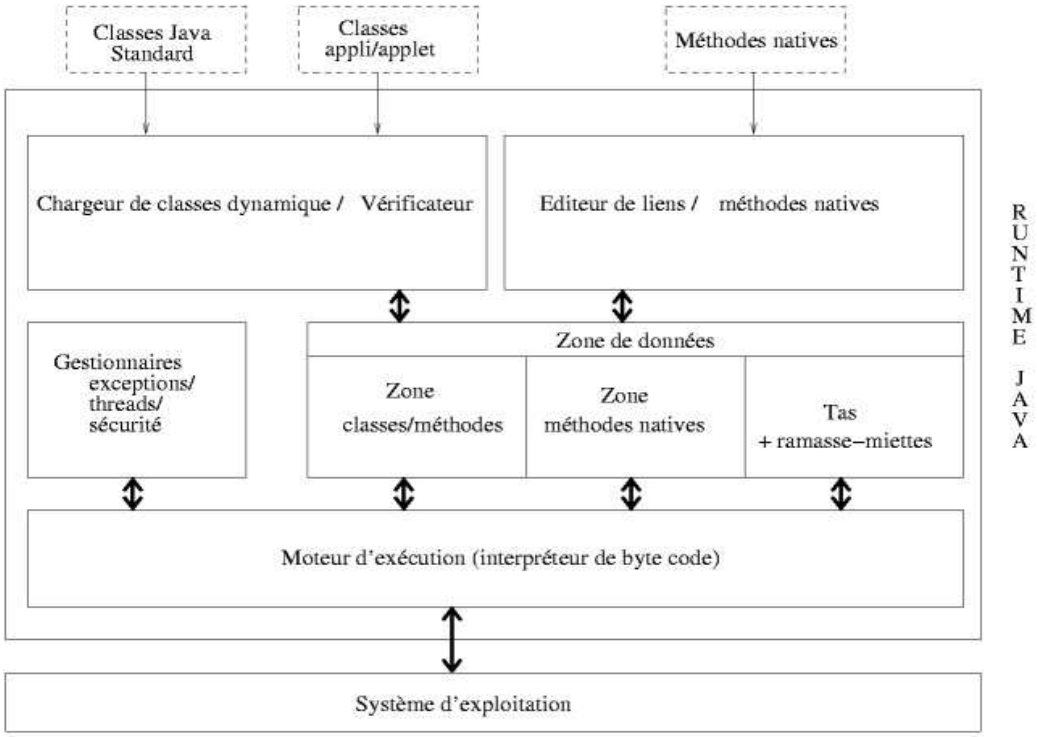

FIG. 4.3 – Vue d'ensemble de l'architecture d'une JVM

Nous détaillons, dans la suite, le rôle des éléments importants dans l'architecture de la machine virtuelle Java, ainsi que la manière dont ils collaborent entre eux.

4.3.1 Chargement, liaison et initialisation des classes

Considérons le programme `Example.java` de la figure 4.6. Après avoir été compilé, celui-ci peut être exécuté par la commande suivante :

```
java Example
```

Ceci suppose que la classe `Example` définisse la méthode `main` ci-après : `public static void main(String[])`. L'exécution de la commande ci-dessus (`java Exemple`) déclenche une série d'étapes, qui sont pour l'essentiel :

- le démarrage d'une instance de la machine virtuelle Java,
- le chargement de la classe `Example`,
- la liaison de cette classe aux autres classes,
- son initialisation,
- puis enfin l'exécution de sa méthode `main`.

Le chargement consiste à chercher le fichier `class` de nom `Example.class`, à le lire puis à incorporer son *bytecode* au sein de la machine virtuelle Java. Cette étape de chargement est laissé au soin du **chargeur de classes** de la machine virtuelle Java.

L'étape de liaison comporte trois phases. Elle commence par une vérification de la classe chargée. Cette phase est importante car la machine virtuelle peut être amené à exécuter du *bytecode* dont elle ignore la provenance, par exemple obtenu *via* le réseau. Cette étape de vérification est prise en charge par le **vérificateur** de la machine virtuelle Java.

La deuxième phase est la préparation, qui crée les champs statiques de la classe, les initialise à leurs valeurs par défaut et met en place certaines structures de données utiles à l'exécution (par exemple, la table des méthodes).

La troisième phase est la résolution qui transforme les références symboliques d'autres types, de leurs champs de leurs méthodes ou de leurs constructeurs en données concrètes. Dans un fichier au format `class`, ces références symboliques sont regroupées au sein de la table `constant_pool` dont nous avons parlée au chapitre précédent. Cette phase peut se produire juste après la vérification ou bien être différée jusqu'à l'exécution, quand une référence symbolique est effectivement utilisée. Dans ce cas, elle provoque le chargement d'autres classes, leur vérification, préparation, résolution, etc. L'étape de résolution est laissée au soin du **gestionnaire de résolution dynamique** de la machine virtuelle.

L'initialisation d'une classe consiste principalement à initialiser ses champs statiques, mais elle requiert que ses super-classes (mais aussi éventuellement ses super-interfaces) aient été initialisées auparavant. Ceci induit le chargement de celles-ci, leur vérification, leur préparation, leur résolution et ainsi de suite.

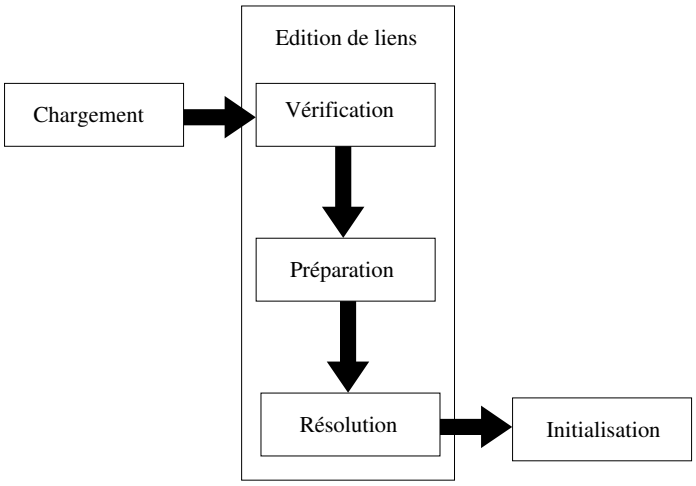

FIG. 4.4 – Chargement d'une classe par la machine virtuelle Java

Lors de l'exécution, la machine virtuelle Java utilise les informations effectivement chargées. Elles sont contenues dans ce que l'on appelle la **zone de données**.

4.3.2 La zone de données

L'espace mémoire de la machine virtuelle Java est composé de plusieurs zones de données (voir figure 4.3) :

- la **zone des méthodes**, qui contient le code des méthodes et des constructeurs mais également les informations sur la structure de chaque classe, et en particulier les informations contenues dans la table nommée `constant_pool` ;
- le **tas** qui contient les instances de classes et les tableaux ;
- la **pile Java**, propre à chaque processus léger, qui contient les cadres d'invocation des méthodes en cours d'exécution (ce sont les *frame* que nous décrivons plus en détails dans la section consacrée au moteur d'exécution) ;
- le **registre pc**, qui est un compteur d'instruction, propre à chaque processus léger et qui désigne l'instruction en cours d'exécution.

Bien que la même zone de données existe dans plusieurs implantations, sa spécification est assez abstraite. Les décisions quand à la manière de l'implanter sont laissées à la discrétion du programmeur. Cependant, dans cette zone de données, le tas possède une place importante.

4.3.2.1 Le tas

Le tas une zone de mémoire qui est partagée par toutes les processus légers démarrés par la machine virtuelle Java. C'est dans cette zone que l'on attribue de la mémoire pour les instances de classes et les tableaux.

Les zones mémoires utilisées ne sont jamais explicitement détruites du tas par le programmeur. En effet, le langage de programmation Java ne permet pas au programmeur de faire figurer explicitement, dans le code source, la libération de mémoire pour les objets créés. La machine virtuelle Java possède les instructions d'allocation de mémoire au sein du tas, utilisées lors de la création d'objets et de tableaux mais elle ne possède aucune instruction permettant l'action inverse, à savoir libérer l'espace mémoire qui a été alloué. De ce fait, la machine virtuelle prend elle-même en charge la libération de mémoire des objets auxquels on ne fait plus référence au sein d'un programme. Elle effectue cela par l'intermédiaire d'un **ramasse-miettes** (*garbage collector*). Celui-ci est chargé de libérer de façon automatique l'espace mémoire alloué pour des objets qui ne sont plus référencés au sein d'une application. Les techniques utilisées pour ce recyclage automatique de la mémoire sont intimement liées à la représentation des objets en mémoire au sein du tas.

4.3.3 Le moteur d'exécution

Le moteur d'exécution gère l'interprétation des instructions de *bytecode* de la machine virtuelle Java. Il manipule également l'exécution des processus légers au sein de celle-ci. Détaillons le déroulement d'une exécution d'un programme Java en terme de pile d'exécution et abordons ensuite les instructions de la machine virtuelle Java.

4.3.3.1 La pile des processus légers

Lors de sa création, chaque processus léger se voit attribuer une pile d'exécution : la *pile Java*. A la différence des piles d'exécution de langages comme le langage C, la *pile Java* n'empile et ne dépile que des piles plus petites : les *frames*. Ainsi, à chaque appel d'une méthode, une nouvelle *frame*, dont la taille maximale précalculée par le compilateur est empilée sur la *pile Java*, et lorsque celle-ci termine son exécution, sa *frame* est dépilée.

L'exemple 4.5 illustre l'incidence de l'invocation de méthodes sur la pile Java d'un processus léger. Dans celui-ci, la méthode `f()` est invoquée. Celle-ci invoque la méthode `g()`, qui elle-même invoque la méthode `h()` : leur *frame* respective est empilée sur la pile Java successivement après chaque appel.

Cet exemple montre aussi les effets de la terminaison de la méthode `h()` sur la pile Java : la *frame* de celle-ci y est dépilée.

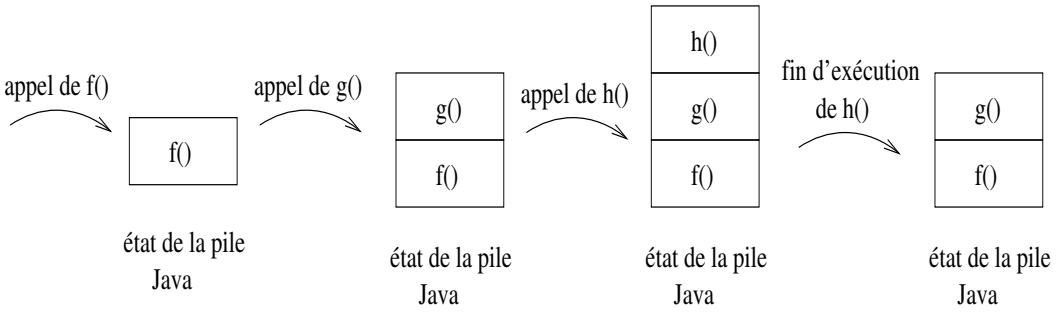

FIG. 4.5 – Manipulation des *frames* par une *pile Java*.

Une *pile Java* ne stocke donc pas directement les données propres à l'exécution d'une méthode, comme par exemple les variables locales ou les valeurs de ses paramètres. Elle délègue le stockage à chacune des *frames*. La pile des *frames* pouvant être implantée par liste chaînée par exemple, sa mémoire n'a pas non plus besoin d'être contiguë, ni même d'avoir une taille fixe.

4.3.3.2 Les frames

Une *frame* est une zone de mémoire qui contient les données locales nécessaires à l'exécution d'une méthode (en particulier ses paramètres, ses variables locales et les résultats partiels de son exécution). Dans son comportement, elle est proche d'une pile d'exécution. Elle est aussi utilisée dans la gestion des valeurs de retour la méthode à laquelle est rattachée, mais aussi de la propagation des exceptions éventuelles. Une *frame* n'est associée qu'à un unique processus léger. Elle ne peut donc faire référence à une autre *frame* d'un autre processus léger. Lors de la terminaison normale d'une méthode (c'est-à-dire sans exceptions), la valeur de retour éventuelle est empilée sur la *frame* de la méthode appelante. La *frame* courante est ensuite dépilée de la *pile Java* puis détruite (*cf.* figure 4.5).

Chaque *frame* se compose de deux parties distinctes :

- un tableau des variables locales,
- une pile d'opérandes.

4.3.3.3 Tableau des variables locales

Le tableau des variables locales d'une *frame* est utilisé pour stocker la valeur de chaque argument d'une méthode, mais aussi celle de ses variables locales. Chacune de ses entrées est capable de contenir la valeur de n'importe quel type défini dans le langage à l'exception des types `double` et `long` qui doivent être contenues au sein de deux entrées. La taille du tableau des variables locales est déterminée à la compilation du programme Java.

Lors d'une invocation d'une méthode de classe, les paramètres sont placés consécutivement dans le tableau à partir de l'indice zéro. Lors de l'invocation d'une méthode d'instance, par contre, l'entrée située à l'indice zéro est utilisée pour contenir la référence de l'objet sur lequel la méthode est invoquée (cette référence correspond au `this` qui est utilisé au sein du programme Java). Les autres paramètres sont, quant à eux, placés consécutivement à partir de l'indice 1.

Pour illustrer l'utilisation des tableaux de variables locales, considérons le programme `Example.java` de la figure 4.6. Il se compose de trois méthodes dont la méthode `main`, qui sera la première à être exécutée par la machine virtuelle Java. Dans cette exemple, la méthode `main` ne déclare que deux variables locales, `value` et `longValue` (lignes 15-16), et les utilise comme paramètres d'appel de `staticMethod` et `instanceMethod`, respectivement méthode de classe (c'est-à-dire statique) et méthode d'instance. Elles effectuent l'affichage des paramètres ainsi que celui d'une variable locale (elle est de type `long` pour `instanceMethod` et de type `int` pour `staticMethod`).

La figure 4.7 illustre l'utilisation des tableaux des variables locales au sein des *frames* allouées pour chacune des méthodes du programme `Example.java`. La disposition de chacune de ses variables (l'ordre dans lequel elles seront empilées sur la piles des opérandes, par exemple) étant dépendante du compilateur, celle que nous avons choisie est la plus intuitive. Cependant, les propos que nous tenons dans la suite sont en accord avec la spécification de la machine virtuelle Java.

Lors de l'exécution de la méthode `main`, la première entrée contient la référence du tableau `args`, qui contient les chaînes de caractères de la ligne de commande. Il est créé par la machine virtuelle et sa référence est passée comme première valeur de l'entrée du tableau des variables locales de la méthode `main`. Les autres entrées du tableau contiennent successivement les valeurs des variables locales `value` et `longValue`, c'est-à-dire 26 et 15000000 (lignes 15 et 16). Cette dernière étant de type `long`, elle est représentée sur deux entrées successives dans ce tableau.

La méthode `staticMethod` est appelée par la méthode `main` avec les arguments "christophe" et avec comme variables locales `value` et `longValue`,

```
1   public class Example {
2     public static void staticMethod(String s, int i, long l) {
3       int val = i*i;
4       System.out.println(val + " " + l);
5       System.out.println(s);
6     }
7
8     public void instanceMethod(String s, int i, long l) {
9       long val = l + i;
10      System.out.println(val);
11      System.out.println(s);
12    }
13
14    public static void main(String[] args) {
15      int value = 26;
16      long longValue = 15000000L;
17
18      //appel de la méthode statique
19      Example.staticMethod("christophe", value, longValue);
20
21      //appel de la méthode d'instance
22      Example example = new Example();
23      example.instanceMethod("christophe", value, longValue);
24    }
25  }
```

FIG. 4.6 – Example.java

de valeurs respectivement 26 et 15000000 (ligne 19). Ainsi, les trois premières entrées du tableau des variables locales de cette nouvelle *frame* contient (voir figure 4.7) :

- la référence de l'instance de la classe `java.lang.String` qui représente la chaîne "christophe",
- la valeur 26,
- la valeur 15000000 stockée sur deux entrées.

Le passage d'arguments entre les méthodes `main` et `instanceMethod` s'effectue de manière similaire, ces arguments étant identiques à la méthode `staticMethod` précédemment appelée. Cependant, `instanceMethod` est une méthode d'instance. Ainsi, la première entrée du tableau des variables lo-

cales de sa *frame* contient la référence de l'objet sur lequel cette méthode est appelée, c'est-à-dire l'objet `example` créé à la ligne 22 du programme de la figure 4.6. Les autres entrées contiennent la valeur des paramètres d'appels de cette méthode, c'est-à-dire : la référence de la chaîne "christophe", 26 et 15000000. Ils y sont stockés comme lors de l'exécution de la méthode `staticMethod` (voir figure 4.7).

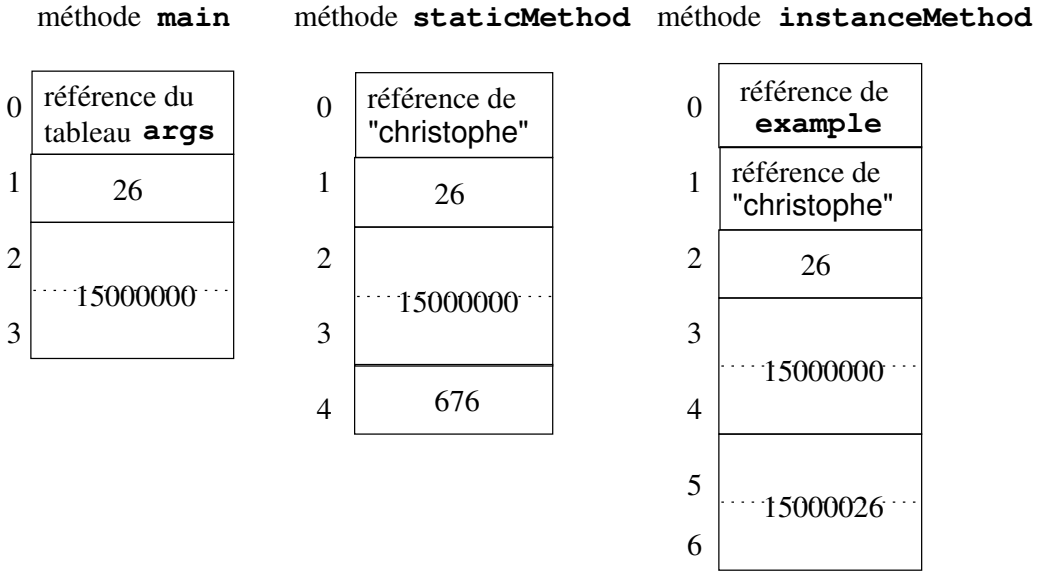

FIG. 4.7 – Détails des tableaux des variables locales pour chaque *frame* des méthodes du programme Example.java

4.3.3.4 Pile des opérandes

Chaque pile d'opérandes est de type LIFO (Last In First Out) et tout comme pour le tableau des variables locales, sa taille est déterminée à la compilation. Elle sert de pile d'exécution pour la méthode. Il faut noter que les paramètres de l'appel n'y sont pas stockés (ils sont dans le tableau des variables locales de la *frame* associée). Elle est utilisée entre autres pour préparer les paramètres à passer à une méthode et pour recevoir les résultats d'une autre méthode. La machine virtuelle y empile également les opérandes des instructions et y dépile leur résultat. Abordons cet aspect en décrivant les instructions de la machine virtuelle Java.

4.3.3.5 Les instructions de la machine virtuelle Java

Chaque instruction de la machine virtuelle Java est codée sur un octet et il en existe environ 200 actuellement. Ce nombre peut paraître important, à première vue. Cependant, plusieurs instructions effectuent les mêmes opérations mais pour des types de données différents (entier, réel, ou encore référence).

Parmi ces instructions, on distingue :
- les instructions qui transfèrent des valeurs entre le tableau des variables locales et la pile des opérande d'une *frame*,
- les instructions arithmétiques qui calculent un résultat qui est typiquement une fonction de deux valeurs de la pile des opérandes, et empile le résultat sur cette pile,
- les instructions de conversion entre les types numériques du langage Java,
- les instructions de création d'objets,
- les instructions d'accès aux variables d'instances ou de classe,
- les instructions de manipulations de la pile des opérandes,
- les instructions de branchements,
- les instructions d'invocations et de retour de méthodes,
- les instructions levant explicitement les exceptions.

Comme nous venons de l'aborder, certaines des instructions de la machine virtuelle Java manipulent la pile des opérandes d'une *frame*. Certaines y empilent des valeurs issues du tableau des variables locales, tandis que d'autres y dépilent une valeur pour la stocker dans une entrée de ce tableau.

Les valeurs contenues par la pile des opérandes et le tableau des variables locales n'étant pas typées, chaque instruction les interprète à sa convenance. Le vérificateur de *bytecode* garantit qu'elles sont correctement typées par les instructions qui composent le fil d'exécution d'une méthode. Il interdit par exemple une séquence d'instructions qui empile un entier et puis interprète le sommet de la pile comme un réel.

Une machine virtuelle Java ne possède pas de registres pour manipuler les valeurs des données intermédiaires. Ainsi, les instructions de la machine virtuelle Java utilisent la pile des opérandes et le tableau des variables locales pour le stockage des données intermédiaires. Cette approche permet de garder un ensemble d'instructions compact et permet de fournir des implantations d'une machine virtuelle sur des architectures ne possédant pas ou peu de registres.

Considérons, par exemple, une *frame* allouée pour une méthode, ainsi que

la séquence d'instructions suivante :

```
iload_0
iload_1
iadd
iload_2
```

Les instructions `iload_0` et `iload_1` empilent respectivement sur sa pile d'opérandes les valeurs contenues dans les entrées d'indice 0 et 1 de son tableau des variables locales, interprétées comme des entiers (`int`). L'instruction `iadd` dépile ces deux valeurs de type `int` et empile le résultat de leur addition, qui lui aussi est de type `int`. L'instruction `istore_2` dépile cette valeur pour la stocker à l'entrée d'indice 2 du tableau des variables locales. Cet exemple illustre la collaboration des deux parties d'une *frame*, mais aussi comment les opérations effectuées peuvent être typées (ici, seul le type `int` est utilisé).

4.3.4 Les autres éléments de l'architecture

Parmi les autres éléments importans de l'architecture de la machine virtuelle, on peut distinguer l'**ordonnanceur des processus légers** et le **gestionnaire de sécurité**.

La spécification de la machine virtuelle Java [LY99] n'impose aucune stratégie d'ordonnancement des processus légers. Elle règlemente uniquement leur accès aux données. Le gestionnaire de sécurité de la machine virtuelle permet également de réglementer l'accès aux ressources de la machine physique sur laquelle la machine virtuelle s'exécute, l'accès aux composants applicatifs et l'accès à certaines fonctionnalités des API.

4.4 Conclusion

Dans ce chapitre, nous avons examiné l'architecture d'une machine virtuelle Java décrite dans la spécification de Sun Microsystems [LY99]. Corosol implante cette spécification, qui guide également son architecture en terme de *composants* (*cf.* chapitre 6).

Troisième partie

Corosol : Une JVM adaptative en Java

Chapitre 5

Propriétés de Corosol

Dans ce chapitre, nous examinons les différentes propriétés de Corosol : les propriétés *statiques* (avant son exécution) et les propriétés *dynamiques* (pendant son exécution).

5.1 Propriétés de notre interprète

Notre objectif est de permettre à une application Java d'évoluer en fonction des besoins au moyen d'une adaptation ou d'une évolution de son interprète, mais aussi par la modification de la représentation de celle-ci.

5.1.1 Propriétés statiques

Exposons tout d'abord les propriétés statiques de notre interprète. Par propriétés statiques, nous entendons toutes les caractéristiques que celui-ci doit posséder avant son exécution. Elles sont de deux sortes :

- modularité au niveau de l'architecture,
- portabilité de l'exécution.

5.1.1.1 Modularité de l'architecture

Nous désirons que notre machine virtuelle possède une architecture où chaque entité fonctionnelle ou de stockage décrit dans la spécification de Sun est représenté par un composant clairement identifiable, pour permettre de la redéfinir facilement. Les dépendances des éléments de cette architecture doivent donc être minimales. Ainsi, le tas, l'ordonnanceur ou encore le chargeur de classes doivent être clairement identifiables dans Corosol. Il doit en être de même pour les différents mécanismes d'exécution comme le mécanisme de résolution d'appel de méthodes.

Bénéficier de cette propriété permet au programmeur dans un premier temps, puis à l'application, dans un second temps (nous examinerons cela dans les sections suivantes), d'identifier et isoler la ou les composantes de l'interprète qu'il faut paramétrer voire remplacer, en fonction des contraintes d'exécution.

5.1.1.2 Redéfinition des composantes de l'architecture avant l'exécution

Nous désirons un interprète dont chacun des éléments de l'architecture puisse être redéfini, dans un premier temps, avant l'exécution de notre machine virtuelle, au moyen d'un fichier de configuration. Celui-ci doit identifier clairement l'implantation associée à un élément de l'architecture. Ainsi suivant le contexte d'exécution de l'application, un choix doit pouvoir être effectué sur l'élément le plus adapté pour les besoins de celle-ci. On appellera par la suite cet élément un *composant* de notre machine virtuelle Java.

5.1.1.3 Conservation de la portabilité de l'exécution

Apporter des modifications à l'interprète Java remet en cause un des attraits principaux de ce langage : la portabilité de l'exécution. Les travaux tournant autour des machines virtuelles aboutissent le plus souvent à des interprètes *ad-hoc*, que ce soit dans l'utilisation de méta-objets comme dans Guaraná ou MetaXa, ou encore pour la réalisation d'une propriété particulière de l'application comme la persistance (voir des outils comme PEVM [LMG00], la machine virtuelle Java de Sun Microsystem qui possède cette propriété).

Par soucis de conservation de la portabilité, Corosol sera exécutée par un autre interprète, qui sera désigné dans la suite par machine virtuelle *hôte*. Ainsi toute modification de Corosol en une machine virtuelle non standard sera exécutée par cet interprète, qui lui restera standard : la portabilité reste assurée.

Une exécution organisée en couches. Corosol est exécutée par une autre machine virtuelle. Son modèle d'exécution est donc organisé en trois couches ou niveaux d'exécution (voir figure 5.1) :

- le niveau applicatif, c'est-à-dire le programme à exécuter,
- le niveau Corosol, qui interprète le niveau applicatif,
- le niveau de la machine virtuelle *hôte*.

Ces différentes notions seront approfondies en particulier au chapitre 7.

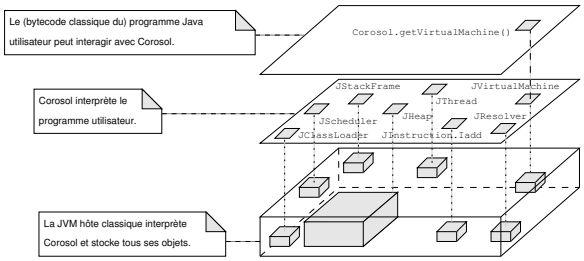

FIG. 5.1 – Organisation en couches de l'exécution de Corosol.

5.1.2 Propriétés dynamiques

Parmi les propriétés de notre interprète, les propriétés dynamiques sont les plus importantes. Elles doivent permettre à l'application de s'observer elle-même par introspection, mais aussi de modifier la sémantique de sa propre exécution. Ces propriétes sont bien sûr liées aux propriétés statiques de notre interprète, en particulier la modularité de son architecture. En effet, l'application doit pouvoir au besoin modifier une composante de celle-ci durant sa propre exécution, et pour cela chaque composante doit être facilement identifiable. Ces propriétés dynamiques de notre interprète sont mises en œuvre par réflexivité.

5.1.2.1 Réflexivité structurelle

Dans un premier temps, notre interprète Java doit pouvoir donner accès à l'application aux différents éléments qu'il utilise pour la représenter, c'est-à-dire la représentation des classes qu'elle manipule, leurs méthodes et leurs champs.

Dans un second temps, notre interprète doit donner accès par intercession à ces mêmes éléments. L'application doit pouvoir modifier les éléments servant à sa propre représentation. Ainsi, notre machine virtuelle doit offrir, *via* ce mécanisme d'intercession, la possibilité d'ajouter, de supprimer ou de renommer les classes de l'application avec leurs méthodes et leurs champs. La granularité de manipulation du code applicatif est poussé jusqu'aux instructions.

Ainsi, *via* la réflexivité, nous désirons un interprète qui donne au programme qu'il exécute la possibilité d'examiner et de modifier les structures utilisées pour sa représentation. Cette machine virtuelle doit aussi fournir à l'application un moyen de modifier la sémantique son exécution.

5.1.2.2 Réflexivité comportementale sur l'interprète

Dans un troisième temps, et toujours par réflexivité, l'application doit pouvoir facilement accéder à n'importe quel composant de son interprète. Elle doit aussi pouvoir redéfinir au besoin chacune d'entre elles. Cela signifie que l'application peut ainsi modifier la sémantique de sa propre exécution. Par défaut, tous les éléments de l'architecture de l'interprète doit être accessible et redéfinissable par le programme exécuté. Cela pose bien entendu des problèmes de sécurité et de sûreté d'exécution, que nous choisissons de ne pas traiter dans un premier temps. Cela signifie que l'application peut mettre en péril sa propre exécution ainsi que celle de son interprète en modifiant la sémantique d'un des composants d'une manière non conforme à la spécification de la machine virtuelle [LY99] de Sun, ou à celle que nous proposons au chapitre suivant.

5.1.2.3 Symbiose

Corosol permet à l'application de manipuler par introspection les composants de la machine virtuelle comme des objets lui appartenant : les objets Java utilisés dans l'implantation de Corosol sont présentés par l'interface d'introspection comme des objets du domaine de l'application. Pour permettre à l'application de redéfinir à la volée des composants de Corosol, l'inverse est également vrai : des objets du domaine de l'application peuvent être utilisés par Corosol pour s'exécuter.

Corosol utilise comme SOUL [WD01] un mécanisme *Up/Down* pour réaliser la symbiose des objets de niveau applicatif et de niveau Corosol.

Chapitre 6

Une architecture à composants

6.1 Introduction

L'architecture à composants de Corosol est présentée dans ce chapitre. Conformément à nos objectifs, chaque unité fonctionnelle ou de stockage qui la compose est clairement identifiable pour faciliter son remplacement avant ou pendant l'exécution, selon les besoins de l'application à interpréter. L'implantation étant réalisée en Java, la portabilité du *bytecode* à exécuter est préservée.

Ce chapitre aborde, dans un premier temps, les aspects généraux de l'architecture de notre machine virtuelle, en définissant la notion de *composant de Corosol* et en expliquant le rôle de chacun d'entre eux. Dans un second temps, il donne une description orientée objets de cette architecture, en spécifiant chaque composant et ses interactions avec les autres par des diagrammes de classes et de séquences.

6.2 Aspects généraux de l'architecture

Dans l'architecture de notre interprète, la machine virtuelle Java est vue comme un *conteneur* de composants, où ceux-ci peuvent être ajoutés, remplacés ou supprimés. Chaque composant possède une ou plusieurs dépendances avec les autres composants de l'architecture. La figure 6.1 montre, par exemple, que le composant C dépend des composants A et D.

Il existe de nombreuses définitions pour les composants, comme celle de Clemens Szyperski [Szy98], de nombreux modèles, en particulier dans le cadre des intergiciels (ou *middleware*) à composants, comme le modèle Fractal [BCS02, BCS] du consortium européen *ObjectWeb*, le modèle de composants CORBA [COR], les EJB [EJB] de Sun Microsystems ou encore

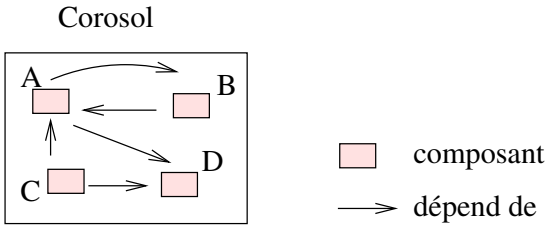

FIG. 6.1 – La machine virtuelle Corosol : un conteneur de composants.

DCOM [NET] de Microsoft.

Selon les besoins et les propriétés définis au chapitre précédent, nous ne nous reposons pas sur ces différents modèles ou définitions. Nous proposons une définition et un modèle plus simple qui répondent à nos besoins. Nous précisons maintenant ce que nous entendons par composant de Corosol.

6.2.1 Composant

La spécification de Sun Microsystems [LY99] définit l'architecture de la machine virtuelle Java en terme d'unités fonctionnelles ou encore d'unités de stockage (*cf.* chapitre 4), ces unités collaborant entre elles : nous les appelons *composants*.

L'architecture de notre interprète Java en est inspirée, mais prend en compte la possibilité de l'ajout ou du remplacement d'un composant, soit avant l'exécution ou soit pendant, sans changer les autres composants. Un composant peut créer d'autres objets durant l'exécution pour son usage propre. De tels objets seront appelés *éléments internes*. Les composants sont ainsi des fabriques abstraites d'éléments internes. Par exemple, lorsqu'il charge le *bytecode* d'une classe, le composant *chargeur de classes* possède la responsabilité de la création de l'élément interne correspondant à celle-ci, mais également des éléments internes représentants ses méthodes et ses champs.

Conformément aux objectifs rappelés au chapitre précédent, les composants de Corosol *et* ses éléments internes pourront être examinés *via* réflexivité depuis l'application. Un composant réifié au niveau de l'application permettra d'accéder à la réification des éléments internes qu'il a créés. Le programme exécuté par Corosol doit modifier ses deux types d'entités pour évoluer ou pour modifier la sémantique de son exécution.

Chaque composant est défini en deux parties distinctes : une partie *abstraite* et une partie *concrète*. À la partie abstraite est associée la spécification abstraite de ce composant et à la partie concrète une implantation effective.

6.2.1.1 Partie abstraite d'un composant

La partie abstraite d'un composant est associée à ce que nous qualifions par sa *spécification abstraite* qui décrit tout d'abord le *type abstrait* du composant. Un composant n'est connu des autres composants de l'architecture que par son type abstrait. Il est ainsi identifié de manière unique par celui-ci au sein du *conteneur de composants*, c'est-à-dire notre machine virtuelle.

La spécification abstraite définit aussi la liste des *services* d'un composant. Un service décrit une action à effectuer par le composant lors de l'exécution de la machine virtuelle. Il est caractérisé par son nom et par le type de ses paramètres. Le composant *chargeur de classes* possède, par exemple, le service nommé *loadClass*, qui à partir du nom d'une classe charge le *bytecode* de celle-ci. Ceci est un point commun avec la spécification des services de composants CORBA *via* le langage IDL [COR] ou celle utilisée par Java lors de l'utilisation du mécanisme d'appel de méthodes distant (RMI) [RMI].

La spécification abstraite détermine aussi les *dépendances statiques* d'un composant. Un service peut être réalisé *via* l'utilisation d'autres composants. Cette collaboration peut être exprimée explicitement si les composants sollicités sont paramètres de ce service. Les dépendances statiques d'un composant sont alors déterminées comme l'ensemble de tous les types de composants qui sont paramètre des services de celui-ci. De même, un composant peut également dépendre statiquement d'éléments internes de l'architecture, si un ou plusieurs d'entre eux sont utilisés pour l'exécution d'un ou plusieurs de ses services.

Les dépendances statiques entre composants sont à minimiser pour éviter un trop grand couplage entre ces derniers dès la rédaction de leur spécification abstraite. Cependant, un composant décrit d'autres relations de dépendance durant l'exécution. Nous les qualifions de *dynamiques*. Elles ne sont pas visibles par l'examen des services d'un composant et sont décrites par l'*implantation concrète* exprimée dans la *partie concrète* du composant que nous examinons ci-après.

6.2.1.2 Partie concrète d'un composant

La *partie concrète* d'un composant représente l'*implantation concrète* de ce composant, c'est-à-dire la mise en œuvre de ses services. Elle décrit les *dépendances dynamiques* de ce composant, c'est-à-dire l'ensemble des types abstraits des composants utilisés pour l'implantation concrète. Lors du remplacement d'un composant, ses dépendances dynamiques peuvent être modifiées car l'ensemble des types abstraits des composants qu'il utilise peut changer. Les dépendances dynamiques des composants qui ne sont pas rem-

placés sont préservées *via* l'usage de *mandataires* (voir paragraphe 6.2.2).

Le type dynamique de l'implantation concrète d'un composant est désigné comme étant son *type concret*. Au sein de Corosol, le type abstrait et le type concret d'un composant sont mis en relation afin de pouvoir rechercher l'implantation concrète de celui-ci à partir de son type abstrait. Cela s'effectue notamment grâce au *référentiel des implantations* et au *gestionnaire des associations*, deux éléments internes de notre machine virtuelle.

Il n'existe qu'une seule implantation concrète par type abstrait de composant au sein de Corosol. Cette gestion des composants est simple et elle répond à nos besoin. Nous utilisons la composition pour faire cohabiter plusieurs implantations concrètes d'un même type abstrait de composant. Par exemple, pour obtenir une machine virtuelle possédant deux tas lors de l'exécution, nous définissons un nouveau type concrèt *DoubleTas* qui est construit par composition de deux implantations concrètes du composant *tas*. RMI [RMI] et CORBA [COR] utilisent une autre solution. Il s'agit d'un service de nommage qui identifie de manière unique (*via* une chaîne de caractères) chaque instance d'objets. Dans nos travaux, cependant, il n'est nécessaire d'identifier que les composants, c'est-à-dire un nombre réduit d'entités. Les éléments internes créés par un composant sont obtenus à partir de celui-ci. Il n'est pas donc pas nécessaire de tous les identifier au sein de l'architecture de Corosol.

6.2.1.3 Les références *externes*

Chaque objet Java est identifié de manière unique par une référence dans le tas de la machine virtuelle. Lors des opérations d'invocation de méthode, par exemple, la référence d'objet appelant doit être stockée sur la *frame* courante (voir chapitre 4). C'est aussi le cas lors de l'exécution des instructions de *bytecode* qui réalisent les accès en lecture et en écriture d'une variable d'instance.

Cependant, le langage Java ne permet pas d'accéder à la représentation des références des objets. En langage C, par exemple, ce sont des adresses représentées sous la forme de nombres. Il existe même une arithmétique associée (l'arithmétique des *pointeurs*). Or, chaque élément de l'architecture de Corosol est un objet Java qui possède une référence dans le tas de la machine virtuelle Java sous-jacente, que nous désignons par machine virtuelle *hôte*. Elles ne peuvent être stockées directement dans un composant ou un élément interne qui serait la pile d'exécution de Corosol.

Pour pallier à cela, nous définissons donc la notion de référence *externe* qui est un identifiant unique attribué à chaque composant et élément interne de Corosol. Il est ainsi possible d'en stocker la valeur au sein d'une *frame* de

Corosol, et plus généralement dans n'importe laquelle de ses zones mémoires définies en Java.

Dans la suite de cette thèse, lorsque nous parlerons d'une référence d'un objet Java, nous préciserons qu'elle est *externe*, si elle est attribuée par Corosol. En l'absence de cette précision, il s'agira de sa *vraie* référence utilisée par la machine virtuelle hôte interprétant Corosol.

6.2.1.4 Gestion des composants

Lors de l'exécution, les composants de Corosol sont créés par l'intermédiaire d'un élément interne particulier appelé *référentiel des implantations*. À partir de l'analyse d'un fichier de configuration rédigé avant l'exécution de la machine virtuelle, il sauvegarde l'ensemble des associations entre le type abstrait d'un composant et son type concret (voir figure 6.2).

Toujours lors de l'exécution, l'élément interne appelé *gestionnaire des associations* établit et sauvegarde l'ensemble des relations entre un type abstrait et le composant de type concret associé. Il communique avec le référentiel des implantations pour déterminer l'implantation concrète d'un composant qu'il faut créer à partir de son type abstrait (voir figure 6.2). Il est utilisé après qu'un composant soit ajouté, remplacé ou supprimé de la machine virtuelle.

Ajouter un composant signifie ajouter un nouveau type abstrait de composant dans l'architecture de Corosol, ainsi que son implantation concrète inconnue des autres composants. Cette opération peut modifier les dépendances statiques des autres composants de l'architecture. Elle définit également les dépendances dynamiques du nouveau composant qui sont encapsulées au sein de son implantation concrète. Après cette opération, si A correspond au type abstrait du nouveau composant et C à son type concret, la nouvelle association (A, C) est ajoutée au sein du gestionnaire des associations.

Remplacer un composant signifie remplacer un composant d'un certain type abstrait par un autre de même type. Cette opération préserve les dépendances statiques entre les composants, mais également les dépendances dynamiques des composants qui ne sont pas remplacés. De nouvelles dépendances dynamiques sont apportées par la nouvelle implantation de composant. Après cette opération, si A correspond au type abstrait du composant à remplacer, C_{old} et C_{new} respectivement à son ancien type concret et à son nouveau, l'association (A, C_{old}) est remplacée par (A, C_{new}) au sein du gestionnaire des associations.

Supprimer un composant signifie supprimer son type abstrait de l'architecture de Corosol. Cette opération modifie les dépendances statiques et dynamiques des composants non supprimés de l'architecture. Après cette opération, si A correspond au type abstrait du composant à supprimer et C à

son type concret, l'association (A, C) est supprimée au sein du gestionnaire des associations.

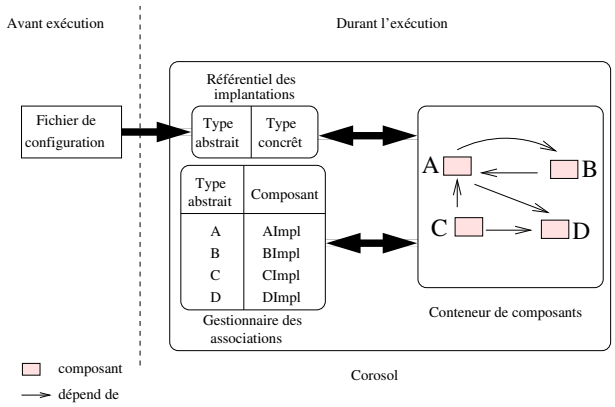

FIG. 6.2 – Associations (*type abstrait / type concret*) de composants.

Afin d'assurer une meilleure gestion des composants et de leurs dépendances, en particulier dans le cas de l'opération *remplacer un composant*, nous avons introduit la notion de mandataire (ou *proxy*) dans l'architecture de notre machine virtuelle Java.

6.2.2 Les mandataires

Lors de l'opération de remplacement des composants, les dépendances dynamiques de ceux qui ne sont pas remplacés doivent être préservées. C'est la raison pour laquelle chaque composant est encapsulé au sein d'un *mandataire* (de l'anglais *proxy*) lors de son ajout au sein de notre machine virtuelle. Ce mandataire possède le même type abstrait que le composant encapsulé (figure 6.3).

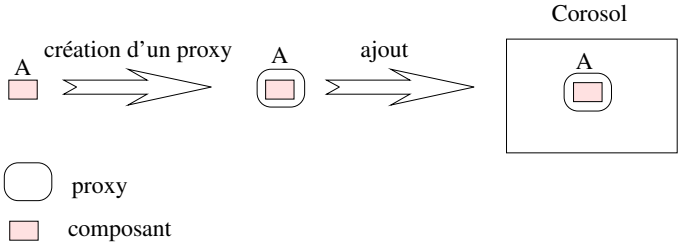

FIG. 6.3 – Création d'un *proxy* lors de l'ajout d'un composant de type A.

L'introduction des *proxys* dans l'architecture de Corosol permet ainsi de faciliter le remplacement à la volée d'un composant par un autre de même type abstrait. En effet, les dépendances dynamiques entre composants rendent difficile cette opération, car lorsqu'un composant est remplacé, sa référence est également modifiée. Tous les composants qui possédaient cette ancienne référence ne peuvent plus accéder aux services de ce composant disparu.

Par exemple, soit un composant de type concret A, qui a besoin d'un composant de type abstrait B et un autre de type abstrait C pour réaliser ses services. Ces deux composants peuvent être remplacés au cours de la vie de la machine virtuelle. Il en est de même pour leur référence respective. La solution est que le composant de type concret A accède aux autres composants B et C *via* des mandataires de mêmes types abstraits B et C. Les références qu'il conserve sont celles de ces derniers. Elles ne seront pas modifiées jusqu'à ce qu'il soit remplacé ou jusqu'à la fin de l'exécution. Elles seront utilisées accéder aux services des composants de types abstraits B et C.

Ainsi, un composant de Corosol ne connait pas les références des composants dont il dépend dynamiquement, mais uniquement les références de leurs mandataires. La figure 6.4 illustre ce qui ce passe par exemple lors du remplacement du composant de type abstrait A. Les différentes flèches représentent les dépendances dynamiques entre tous les composants de Corosol : ils sont au nombre de quatre et sont respectivement de types abstraits A, B, C et D. Les composants qui dépendent de A, à savoir B et C, n'ont pas vu leurs dépendances dynamiques se modifier (sur le dessin, les flèches issues de B et C sont identiques). Seules les dépendances dynamiques de A l'ont été. Dans l'ancienne implantation de A, ce composant dépendait des types abstraites B, C et D : trois flèches étaient issues de A vers ces trois composants. Dans sa nouvelle implantation, le composant de type abstrait A ne dépend plus dynamiquement que de B et D : la flèche issue de A et dirigée vers C a désormais disparue.

6.2.3 Les *méta-classes*

Une *méta-classe* est une classe dont les instances sont elles-mêmes des classes. En Java, la machine virtuelle est à l'origine de leur création et les représente par le type `java.lang.Class`. Il n'existe qu'une instance par type utilisé durant l'exécution. En particulier, il en existe une pour chaque type primitif. Par exemple, la méta-classe du type `Thread` et celle du type primitif `float` sont respectivement `Thread.class` et `float.class`.

Une méta-classe permet d'obtenir les descriptions des méthodes et des

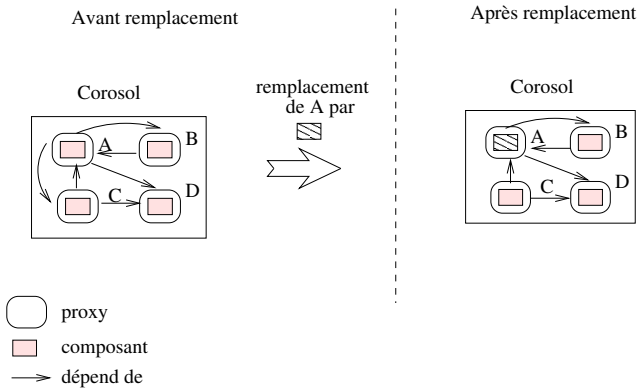

FIG. 6.4 – Préservation des dépendances dynamiques lors du remplacement.

champs de la classe qu'elle désigne. En Java, elle permet d'obtenir la réification des informations contenues au sein du code compilé de cette classe. Par exemple, la méta-classe `Thread.class` permet d'obtenir les méthodes et les champs de la classe `Thread` respectivement sous la forme d'objets `Method` et `Field` du paquetage `java.lang.reflect`.

L'architecture de Corosol définit également des méta-classes. Ce sont des éléments internes qui participent implicitement à la réalisation de l'exécution.

Parmi toutes les instructions de la machine virtuelle Java, beaucoup manipulent les *frames* selon un type précis. Elles réalisent par exemple le transfert des valeurs entre la pile d'exécution et le tas de la machine virtuelle ou entre la pile des opérandes et le tableau des variables locales d'une *frame*.

Par exemple les instructions concernées dont le nom commence par la lettre *i* n'effectuent leurs opérations que sur des valeurs de type `int`. Elles vont dépiler, empiler, stocker, lire d'une *frame* des valeurs de ce type. La nomenclature de ces instructions permet de déterminer le type de la valeur attendue, soit au sommet de la pile des opérandes de la *frame*, ou soit au sein d'une des entrées de son tableau des variables locales.

Corosol définit deux sortes de méta-classes :

- les méta-classes associées aux types primitifs, définies par le programmeur de Corosol,
- les méta-classes associées aux classes des objets, définies automatiquement par Corosol lors du chargement du code compilé associées à ces classes.

Les méta-classes de Corosol associées aux types primitifs permettent de manipuler la pile des opérandes des *frames* selon le type qu'elles représentent.

Par exemple, la méta-classe associée au type primitif `double` sait y dépiler une valeur de type `double`. Elle sait également comment y empiler une valeur de ce type.

Les méta-classes de Corosol associées aux types des objets permettent d'effectuer les mêmes opérations mais sur les objets. Elles vont permettre en particulier de transformer certains objets durant l'exécution (voir chapitre 7).

Les méta-classes de mandataires, invisibles à l'application, sont générées par Corosol et utilisées pour créer les instances de mandataires durant l'exécution.

6.3 Création et initialisation des composants

Le *référentiel des implantations* est le premier objet Java à être créé. Il permet d'instancier par la suite le *conteneur de composants* de Corosol qui acceuillera les autres composants de Corosol.

À l'ajout d'un composant nouvellement instancié par le *référentiel des implantations*, un mandataire est créé pour lui. Chacun des composants dont il dépend dynamiquement ainsi que leurs mandataires respectifs sont créés récursivement par le même procédé. Comme il n'existe qu'un composant pour un type abstrait donné, il n'existe qu'un mandataire qui lui est associé. Les cycles sont ainsi évités.

6.4 Composition de l'architecture

L'architecture de notre interprète est inspirée de la spécification de la machine virtuelle Java [LY99] de Suns Microsystems. Corosol est entièrement écrite en Java. Elle est donc interprétée par une autre machine virtuelle sous-jacente que nous avons précédemment désignée par *hôte*. Les différents composants de l'architecture sont ainsi présentés par des entités Java. Nous représenterons souvent leur organisation et leur fonctionnement par des diagrammes de classes ou de séquences. La figure 6.5 en donne une vue d'ensemble. Elle y présente les composants et tous les autres éléments internes de Corosol.

Nous présentons, dans un premier temps, les éléments de l'architecture de gestion des composants de Corosol puis, dans un second temps ceux l'architecture standard d'une machine virtuelle Java.

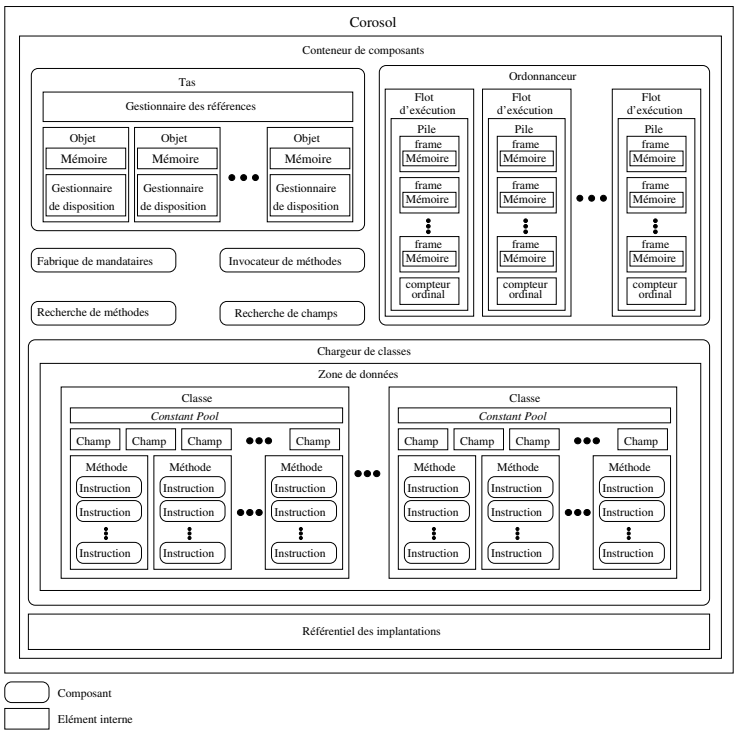

FIG. 6.5 – Vue d'ensemble de l'architecture de Corosol.

6.4.1 Composants de l'architecture de gestion

L'architecture de Corosol se compose d'éléments qui ne se retrouvent pas dans l'implantation classique d'une machine virtuelle. Leur présence à pour objectif d'assurer la gestion des composants de Corosol. C'est cette architecture de gestion que nous décrivons à présent.

6.4.1.1 Les éléments internes `JObject`

Lors de l'exécution d'un programme par Corosol, deux types d'objets peuvent être créés :

- des objets de l'application, qui sont alloués au sein du tas de Corosol,
- des éléments internes (voire des composants), qui doivent résider au sein du tas de la machine virtuelle qui interprète Corosol.

Afin de pouvoir différencier ces deux types d'objets lors de l'interprétation du programme, on définit le type `JObject` comme super-type commun à tous les éléments internes de Corosol, composants compris. Un élément interne participe au fonctionnement de l'exécution de Corosol et n'est pas

spécifiquement destiné à être créé ou remplacé directement durant l'exécution, mais plutôt par l'intermédiaire d'un composant. Cependant, chaque élément interne peut être réifié durant l'exécution. C'est le cas par exemple des méta-classes.

Le type `JObject` ne fait qu'identifier n'importe lequel des éléments de l'architecture de Corosol. Il ne définit donc aucune opération. À l'exemple de l'interface `java.lang.Serializable` qui étiquette chacune des classes sérialisables, chacun des nouveaux types Java possédant ce super-type est reconnu comme faisant partie de l'implantation de notre interprète. Ainsi, si la classe de l'objet à instancier est une sous-classe de `JObject` alors cet objet est un élément interne et doit être alloué au sein de la machine virtuelle *hôte*.

6.4.1.2 Les composants `JVMComponent`

Corosol est une machine virtuelle qui dispose d'un *conteneur de composants* où chacun d'entre eux est manipulé par son type abstrait, indépendemment de son type concret. Tout comme les éléments internes de l'architecture, il est nécessaire de leur définir un super-type commun : le type `JVMComponent`.

Un composant est aussi un élément interne de Corosol, car il fait partie de son implantation. `JVMComponent` est donc un sous-type de `JObject`. De plus, un *composant* pouvant être ajouté, remplacé ou supprimé, le type `JVMComponent` définit les opérations suivantes, communes à tous les composants :

`void configure(JVirtualMachine jvm)` : cette méthode permet de lier le composant aux autres composants en établissant ses dépendances dynamiques ; les références des autres composants sont obtenues *via* le conteneur de composants de type abstrait `JVirtualMachine` sur lequel nous reviendrons dans la suite ;

`Class getComponentClass()` : cette méthode retourne le type *abstrait* du composant ; ce type est celui par lequel un composant est connu des autres au sein de l'architecture ;

`void replace(JVMComponent component)` : cette méthode effectue toutes les opérations nécessaires au transfert d'état entre ce composant et un autre de même type abstrait désigné par `component` ; elle est exploitée par le conteneur de composants lors de l'exécution en cas du remplacement de l'un d'entre eux.

6.4.1.3 Les mandataires `JProxy`

L'architecture de Corosol utilise le modèle de conception des mandataires afin de faciliter la réalisation des opérations de remplacement de composants.

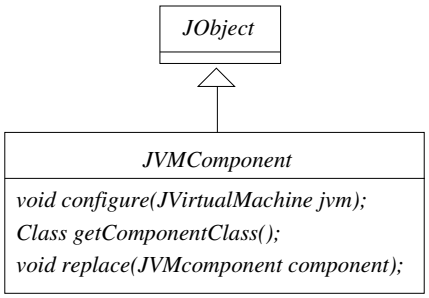

FIG. 6.6 – Composants et éléments internes.

Un mandataire est chargé d'encapsuler un composant Corosol. Un composant donné ne possède pas directement les références des autres composants, sachant qu'elles peuvent être modifiées en cas de remplacement, mais il possède celles de leurs mandataires.

Au cours de l'exécution, un mandataire est une instance d'une classe de mandataire, créée à la volée par la *fabrique abstraite de mandataires*, élément interne de Corosol de type `JProxyFactory`. Pour chaque type abstrait de composant, elle créée par génération de *bytecode* (voir 3.5 et le chapitre 8), une classe de mandataire particulière dont le super-type est `JProxy`. À l'exemple du rôle du type `JObject`, `JProxy` identifie un mandataire dans l'architecture de Corosol. Il possède les opérations suivantes :

`Object getObject()` : cette méthode retourne l'objet encapsulé par le mandataire ; dans le cas d'un mandataire de composant, il doit être un composant.

`void setObject(Object o)` : cette méthode attribut l'objet spécifié comme nouvel objet encapsulé par le mandataire ; dans le cas d'un mandataire de composant, `o` désigne aussi un composant et `setObjet` est utilisée lors du remplacement de composant.

La figure 6.7 illustre la construction de la classe `CProxy` qui est la classe des composants de type abstrait `C`. Comme le montrera le chapitre 7, elle possède `JProxy` et `C` comme super-type et possède une variable d'instance de type `C` qui représente le composant à encapsuler. Ce même chapitre explicitera un second type de mandataire utilisé durant l'exécution. L'objet encapsulé alors ne sera pas un composant. C'est la raison pour laquelle le type `JVMComponent` n'apparaît pas dans le profil des deux méthodes de `JProxy` : seul le type `Object`, commun à tous les types Java, est mentionné.

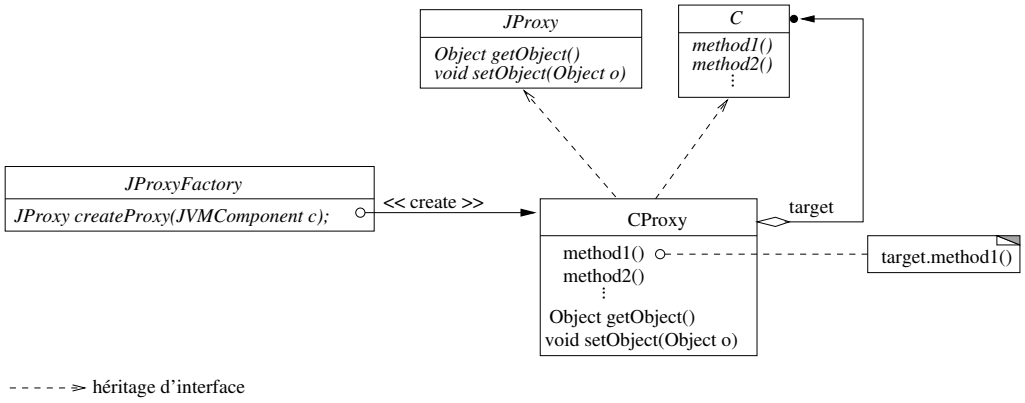

FIG. 6.7 – Les mandataires de composants.

6.4.1.4 Le référentiel des implantations `JImplementationRepository`

Le *référentiel des implantations* est le composant de Corosol qui établit la liaison entre le type abstrait d'un composant et son type concret. Avant le démarrage de Corosol, il détermine ces associations par la lecture d'un fichier de configuration. Durant l'exécution, il spécifie au conteneur de composants de Corosol quelles sont les implantations concrètes à créer.

Le référentiel des implantations joue ainsi le rôle de fabrique abstraite de composants. Mais ce rôle n'est pas exclusif. Il est aussi utilisé pour la création d'autres éléments internes de Corosol, comme les méta-classes que nous décrivons plus loin. Il est également utilisé comme fabrique abstraite de mandataires.

Le type `JImplementationRepository` implantant l'interface `JProxyFactory`, représente le référentiel des implantations de Corosol. La méthode suivante est l'une de ses opérations les plus importantes :

`public Object create(Class abstractType)` : cette méthode recherche le type concret d'un composant dont le type abstrait est désigné par `abstractType`, puis en retourne une nouvelle instance ; bien que cela ne soit pas obligatoire, la même instance peut être délivrée plusieurs fois, si un système de cache d'objets est implanté.

6.4.1.5 Le conteneur de composants `JVirtualMachine`

`JVirtualMachine` est le type abstrait qui représente à la fois le *conteneur de composants* et le *gestionnaire des associations* de Corosol. Il communique avec la fabrique abstraite de mandataires pour les encapsuler avant leur in-

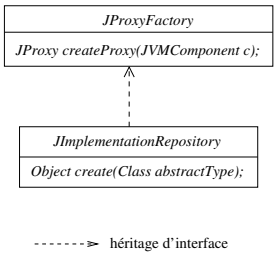

FIG. 6.8 – Le référentiel des implantations.

tégration au sein de la machine virtuelle. Il collabore étroitement avec le réferentiel des implantations lors de l'ajout d'un nouveau composant.

Chaque composant n'étant connu des autres que par l'intermédiaire de son type abstrait, c'est aussi par celui-ci qu'il est recherché, ajouté ou remplacé dans ce conteneur. Il est important de noter qu'au sein de notre machine virtuelle, il n'existe qu'un unique composant pour un type abstrait donné, afin de simplifier la gestion des composants. Le type abstrait `JVirtualMachine`, qui représente le *conteneur de composants*, définit les opérations suivantes :

`JProxy addComponent(JVMComponent component)` :
cette méthode ajoute le nouveau composant `component` au sein de Corosol, en l'encapsulant au sein d'un nouveau mandataire qui possède le même type abstrait que ce nouveau composant, et qui est ensuite retourné par cette méthode (voir figure 6.9) ; l'implantation par défaut de `addComponent` demande au *gestionnaire des associations* d'ajouter la nouvelle relation $(abstractType, component)$, $abstractType$ désignant le type abstrait du nouveau composant, obtenu *via* l'appel de méthode `component.getComponentClass()` ;

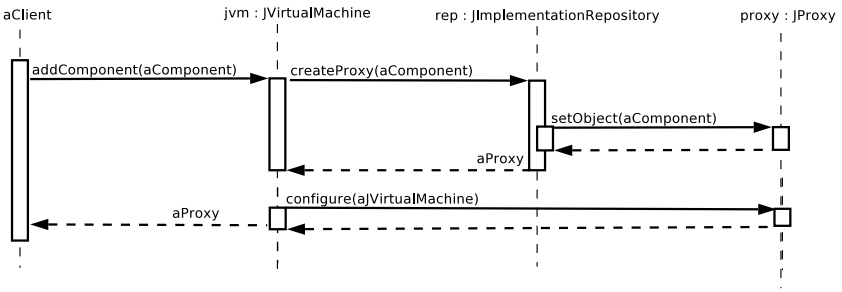

FIG. 6.9 – Ajout d'un nouveau composant.

`JProxy getComponent(Class abstractType)` :
cette méthode retourne le mandataire correspondant au composant

dont le type abstrait est spécifié par la classe `abstractType` ; si ce composant n'existe pas, elle initie les opérations permettant de l'ajouter au sein de ce conteneur ; ainsi par une requête au référentiel des implantations, le type concret du nouveau composant est déterminé puis instancié ; le conteneur l'encapsule ensuite au sein d'un nouveau mandataire de même type abstrait, qu'il retournera (voir figure 6.10) ; dans le cas où il est nécessaire de créer un nouveau composant (appelons-le `component`), l'implantation par défaut de la méthode `getComponent` demande au gestionnaire des associations d'ajouter la nouvelle relation $(abstractType, component)$.

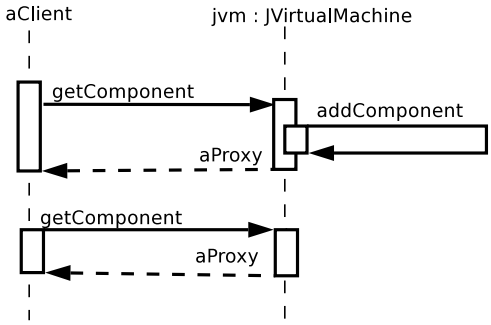

FIG. 6.10 – Consultation d'un composant.

`void replaceComponent(JVMComponent component)` : cette méthode remplace par le nouveau composant désigné par `component` le composant dont le type abstrait est décrit par `component.getComponentClass()` ; `component` est ensuite encapsulé par le mandataire de l'ancien composant remplacé *via* la méthode `setObject` du type `JProxy` (voir figure 6.11) ; la méthode `replaceComponent` demande au gestionnaire des associations de remplacer la relation $(abstractType, oldComponent)$, `oldComponent` représentant l'ancien composant et `abstractType` le résultat de `component.getComponentClass()`, par la nouvelle relation $(component.getComponentClass(), component)$;

`boolean containsComponent(Class abstractType)` : cette méthode détermine si le composant, dont le type abstrait est précisé par la classe `abstractType`, existe au sein du conteneur de composants ;

`void startUp(String className, String[] args)` : cette méthode construit la machine virtuelle Corosol par assemblage des composants et démarre ensuite l'exécution de la classe spécifiée en fonction des paramètres de la ligne de commande désignée par `args` ; l'ensemble des

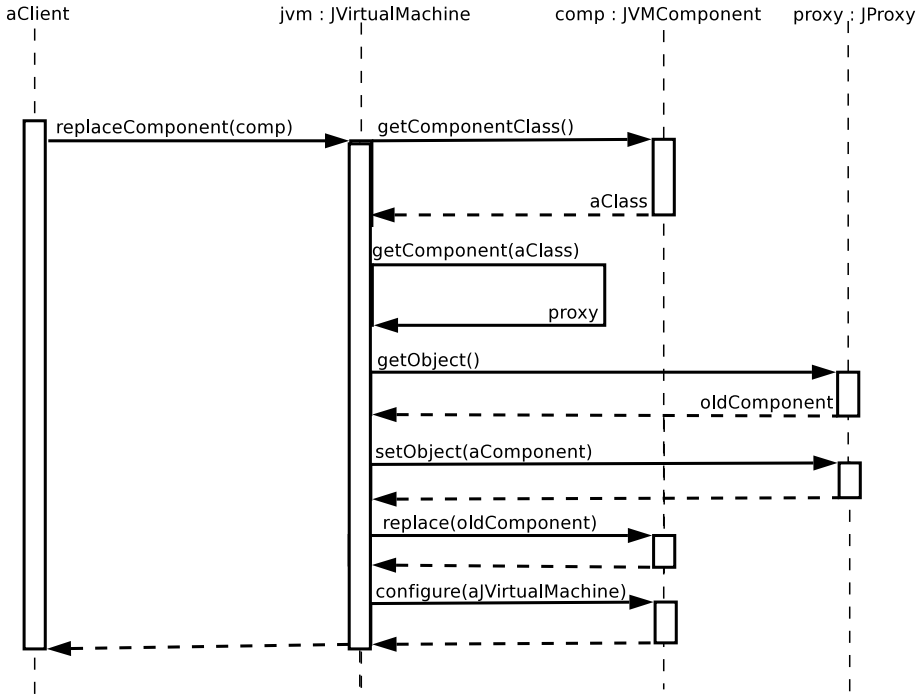

FIG. 6.11 – Remplacement d'un composant.

types concrets de composants à instancier est déterminé *via* des interrogations au référentiel des implantations qui est le premier à être créé ; par exemple dans l'implantation par défaut, les composants et éléments internes à être explicitement instanciés sont les suivants et dans cet ordre :

1. le composant *chargeur de classes* est instancié ; il effectue le chargement de la classe de l'application ;
2. le composant *recherche de méthode* est créé pour la recherche la méthode `main` dans cette classe ;
3. un fil d'exécution associé à l'exécution de la méthode `main` est ensuite construit : il initie la création d'une *frame* pour cette méthode ;
4. le composant *gestionnaire des références* est instancié ; il permet d'associer une référence externe au tableau de chaînes de caractères représentant les arguments de la ligne de commande ; cette référence externe est ensuite empilée sur la *frame* de la méthode `main` ;
5. le composant *ordonnanceur* est créé, le fil d'exécution précédem-

ment créé y est ajouté et la boucle d'exécution de Corosol commence.

L'ajout d'un nouveau composant dans Corosol provoque la création des composants dont il dépend dynamiquement. La création du composant *tas* est par exemple initiée par celle du composant *gestionnaire des références*.

6.4.2 Composants standards

Nous présentons maintenant les composants de Corosol qui sont définis au sein de n'importe quelle machine virtuelle Java et décrits par la spécification de Sun [LY99].

6.4.2.1 Le modèle de mémoire de Corosol

La spécification de Sun décompose la mémoire de la machine virtuelle Java en plusieurs parties, parmi lesquelles le tas, les piles des fils d'exécution ou celles exécutant le code natif, ou encore la zone des méthodes (voir chapitre 6). Cependant, elle n'impose aucune représentation particulière des objets, ni ne précise l'algorithme de ramasse-miettes à utiliser.

Nous proposons ainsi un modèle de mémoire simple et centralisé pour Corosol où le composant *tas* représente *toute* la mémoire de Corosol. Chaque zone de mémoire y est allouée, tant les *frames* que les zones allouées pour les objets. En conséquences, toutes les portions de mémoire utilisées par un programme ont la même représentation. Ajouter un second composan *tas* implique ajouter une seconde zone de mémoire à Corosol qui implante sa propre représentation des données.

La mémoire de notre machine virtuelle est ainsi une séquence d'octets, accessibles en lecture et en écriture *via* un indice de position, représentée par le composant *tas*. Il assure l'encodage et le décodage de la valeur de chacun des types primitifs de Java. Il détermine ainsi le nombre d'octets utilisés pour la représentation de chacun d'entre eux.

Au cours de l'exécution, les données de chaque objet créé sont contenues dans une zone de mémoire réservée au sein du tas. Elle est localisée par l'indice de position de son premier octet. Cette *position absolue* est calculée par le composant *allocateur*.

De même, les indices de position des champs d'une instance de classe et celles des entrées d'un tableau sont calculées par un *gestionnaire de disposition*. C'est un élément interne de Corosol et il en existe exactement un pour chaque classe de l'application. Ces indices de positions sont *relatifs* à la position *absolue* de la zone mémoire allouée pour les instances ou les tableaux.

Leur calcul est effectué en fonction de la taille de chaque type primitif au sein du *tas* (voir figure 6.12).

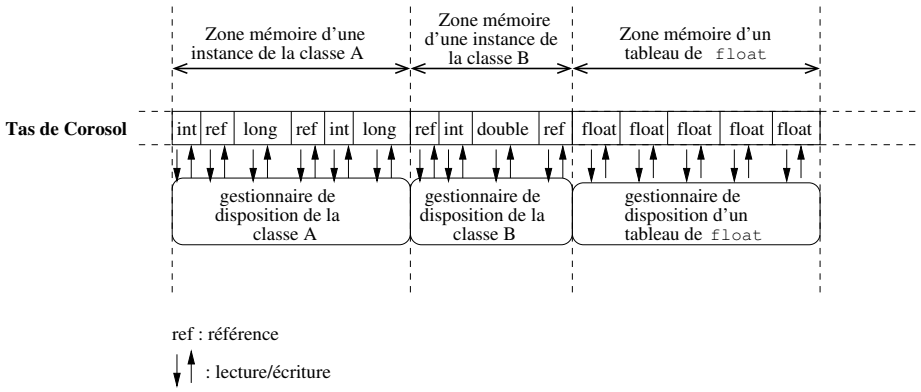

FIG. 6.12 – Allocation des objets dans le tas de Corosol

Chaque fil d'exécution de Corosol possède une *pile Java* (voir chapitre 4) dont la mémoire est une zone allouée au sein du *tas*. Elle y constitue une zone délimitée dont la position absolue est déterminée *via* le composant *allocateur*. Elle se décompose en *frames*, dont les positions dans la mémoire allouée sont calculées relativement à sa position absolue (voir figure 6.13).

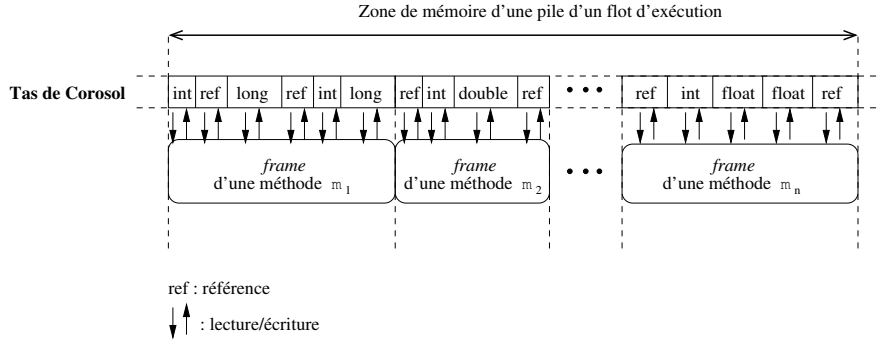

FIG. 6.13 – Allocation des *piles Java* dans le tas de Corosol.

Notre modèle de mémoire sépare l'allocation des objets de la gestion de leur référence externe, car tous les objets créés utilisés par l'application se sont pas tous alloués physiquement au sein du *tas* de Corosol, comme dans le cas particulier de ses composants. Ce point important sera développé au chapitre 7. C'est la raison pour laquelle nous choisissons de mettre en relation, non une référence externe et la position mémoire d'un objet utilisé par l'application, mais plutôt une référence externe et la *représentation Java* de

cet objet : les objets de l'application sont donc ainsi représentés par d'autres objets de la couche inférieure (Corosol).

Cette gestion des références est confiée au composant *gestionnaire des références*. Il attribue une référence externe à chaque objet utilisé par l'application (y compris les composants, voir aussi chapitre 7). Il sauvegarde ainsi l'association entre un objet `o` et sa référence externe, ou plus exactement, l'association entre l'objet qui représente `o` dans la machine virtuelle *hôte* et sa référence externe (figure 6.14). Chaque objet possède ainsi la même référence externe jusqu'à l'éventuel remplacement du composant *gestionnaire des références*.

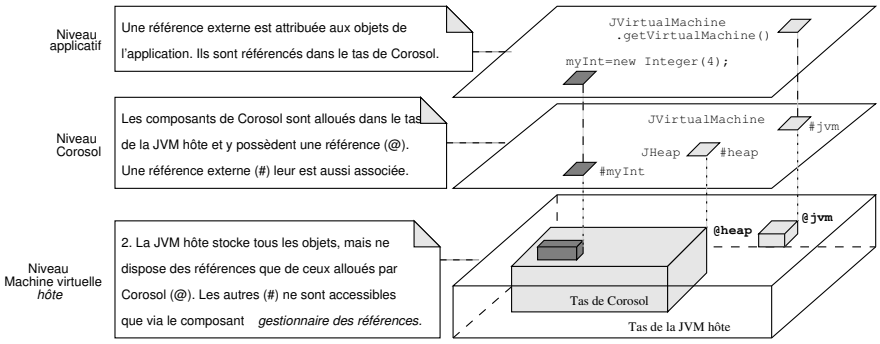

FIG. 6.14 – Références *versus* références externes

Par souci de simplicité, le modèle de mémoire de Corosol n'intègre pas encore les caches mémoire des fils d'exécution. Il en est de même pour les moniteurs.

Discussion sur la représentation des types primitifs Bien que la spécification de la machine virtuelle impose une *valeur* minimale et maximale (ainsi qu'une valeur d'initialisation) pour chacun des types primitifs, elle n'impose en revanche aucune restriction quant à la *représentation* de chacun d'entre eux au sein du tas. Dès lors, une implantation de tas pour la machine virtuelle peut décider d'allouer tous ses types primitifs sur un même nombre d'octets par exemple. Cependant, les valeurs lues au sein de ce tas doivent correspondre aux valeurs prévues dans la spécification.

Les valeurs encodées dans le composant *tas* sont de la taille imposée par Java et possèdent une valeur maximale et minimale. Cependant, le nombre d'octets utilisés par le tas de Corosol pour l'encodage des valeurs primitives peut être supérieure à celle prévue par la spécification de la machine virtuelle [LY99]. Cela signifie que les valeurs stockées par le composant *tas*

peuvent être en théorie plus grande (ou plus petite) que celles imposées par Java. Mais, pour des raisons de simplicité, Corosol effectue les différents calculs à l'aide des types primitifs fournis par Java, même si leur représentation peut être plus petite que celle permise par le tas de Corosol.

Ainsi, les valeurs extraites ou inscrites dans composant *tas* auront la taille imposée par la spécification de Sun et seront de la forme des types primitifs de Java, même si l'encodage des valeurs primitives qu'il utilise permet de représenter des nombres plus grands ou plus petit (voir figure 6.15).

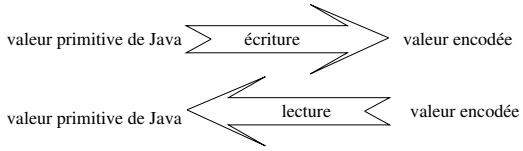

FIG. 6.15 – lecture / écriture dans le composant *tas*

Il est tout à fait possible d'avoir de représenter les valeurs primitives par des tableaux d'octets contenant leur encodage, afin d'effectuer les différents calculs, mais aussi leurs transferts vers ou depuis le composant *tas*. Un autre moyen serait également d'objectiser chaque valeur primitive. Bien que ces solutions soient possibles à mettre en œuvre, nous ne les avons pas choisies pour des raisons de simplicités et avons préféré la limitation de représentation imposée par la spécification de la machine virtuelle.

Examinons maintenant les différents types abstraits des composants intervenant dans le modèle de mémoire de Corosol.

Le composant *tas* `JHeap`**.** Le composant *tas*, qui représente la mémoire de Corosol, possède le type abstrait `JHeap`. Il encode et décode la valeur de chaque type primitif (les références ne sont pas considérées comme un type primitif) de Java respectivement selon son type, *via* les services suivants :

- *Type* `read`*Type*`(int index)`, qui effectue le décodage des octets d'une zone de mémoire commençant à la position absolue `index` et en retourne le résultat sous la forme d'une valeur de type primitif *Type* ;
- `void write`*Type*`(int index,` *Type* `value)`, qui effectue l'encodage de la valeur *value*, dont le type primitif est *Type*, puis l'écrit au sein de la mémoire à partir de la position absolue `index`.

La taille des données encodées et décodées dépend de l'implantation de `JHeap`. Par exemple, dans son implantation par défaut, la valeur d'un entier de type `int` est représentée sur quatre octets au format *big indian*. Les

deux méthodes `int readInt(int index)` et `void writeInt(int index, int value)` permettent respectivement de lire et d'écrire une valeur de ce type à partir de l'`index`-ième octet du tas. Lors de la lecture, quatre octets consécutifs sont décodés et représentés sous la forme d'une valeur de type `int`, et lors de l'écriture, une valeur de ce type est encodée sous la forme de quatre octets consécutifs au format *big indian*.

Notre modèle de mémoire n'imposant pas de représentation particulière pour les références externes, il n'existe pas non plus de type associé à celles-ci. De plus, la valeur de chaque référence externe est mise en relation avec la représentation en Java de l'objet de l'application qu'elle désigne. Tout cela se reflète dans la liste des services du composant *tas*, qui dispose en plus des deux suivants :

- `JHeapObject readReference(int index)` effectue le décodage de la référence externe contenue à partir du `index`-ième octet, puis retourne l'objet correspondant *via* une interrogation au composant *gestionnaire des références* ; cet objet est la représentation en Java d'une instance de classe ou d'un tableau de l'application, de type abstrait `JHeapObject` ;

- `void writeReference(int index, JHeapObject object)`, *via* une interrogation au composant *gestionnaire des références*, détermine la référence externe associée à `object`, qui est la représentation en Java d'un objet de l'application, en effectue l'encodage et écrit celui-ci à partir du `index`-ième octet du *tas*.

Ainsi, les instances de type `JHeapObject`, qui est utilisé dans les services `readReference` et `writeReference`, représentent dans la couche Corosol les objets de l'application qu'elle exécute. `JHeapObject` est un sous-type de `JObject`, comme le montre la figure 6.16. Elle illustre également les sous-types suivants de `JHeapObject` :

- le type `JClassInstance` correspond à la représentation en Java d'une instance de classe de l'application,
- le type `JArray`, à celle d'un tableau.

Une des motivations de l'utilisation des éléments internes `JHeapObject` est donc de masquer la représentation des références externes aux composants utilisant `JHeap` et d'associer celles-ci à un objet Java plutôt qu'à une position dans le tas. Le chapitre 7 développera une autre raison centrée sur l'origine des objets utilisés par l'application : comme nous l'avons brièvement mentionné précédemment, ils peuvent aussi désigner des composants ou d'autres éléments internes de Corosol. Les méthodes de `JHeapObject` montrée à la

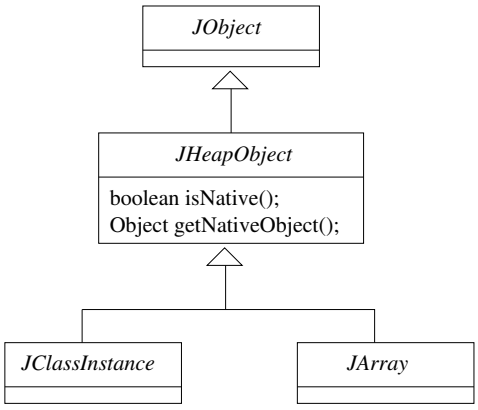

FIG. 6.16 – La représentation des objets dans Corosol.

figure 6.16 y seront également détaillée.

L'implantation des services d'encodage et de décodage des références reposant sur le composant *gestionnaire des références*, nous le décrivons maintenant.

Le composant *gestionnaire des références* `JReferenceManager`**.** Le type abstrait `JReferenceManager` représente le composant *gestionnaire des références* de Corosol. Il possède la responsabilité d'attribuer une référence externe à chaque objet créé au sein de l'application. Il conserve également l'historique des associations entre chaque objet et sa référence. Ces différentes opérations se font par le service suivant :

- `assignReference(JHeapObject object)`, qui attribue une référence à l'objet du tas désigné par `object` ; si `refValue` est la référence attribuée à *object*, alors la nouvelle association (`refValue`, `object`) est ajoutée au sein du composant *gestionnaire des références*.

L'encodage et le décodage des valeurs primitives s'effectuent toutes *via* le composant *tas*. Cependant, il ne possède pas de service permettant d'effectuer *directement* ces opérations sur les références. Aussi, *afin de ne pas imposer un format particulier* pour les références externes, leur lecture et leur écriture est obligatoirement implantée à l'aide des autres services du composant *tas*. Par exemple, dans son implantation par défaut, les références sont encodées et décodées sous la forme de valeurs entières de type `int` *via* l'utilisation des services `readInt` et `writeInt`. C'est le composant *gestionnaire des références* qui possède la responsabilité de *choisir* leur format d'encodage et de codage dans le tas. Il dispose ainsi des deux services suivants :

- `JHeapObject readReference(int index)`, qui effectue le décodage de la référence externe contenue à partir du `index`-ième octet *via* un des services du composant *tas*, puis retourne l'objet `JHeapObject` correspondant ;

- `void writeReference(int index, JHeapObject object)`, qui réalise la recherche de la référence externe de l'objet `object` correspondant, puis l'encode au sein de la mémoire, à partir du `index`-ième octet, *via* un des services du composant *tas*.

La figure 6.17 illustre les appels successifs de l'implantation par défaut de la lecture d'une référence externe dans le composant *tas*. De cet exemple, son service `readReference` sollicite le service du même nom sur le composant *gestionnaire des références*, qui effectue le décodage par l'intermédiaire du service `readInt`. La valeur obtenue lui permet alors de déterminer quel est l'objet associé à cette référence externe, le résultat du service `readReference` du composant *tas*.

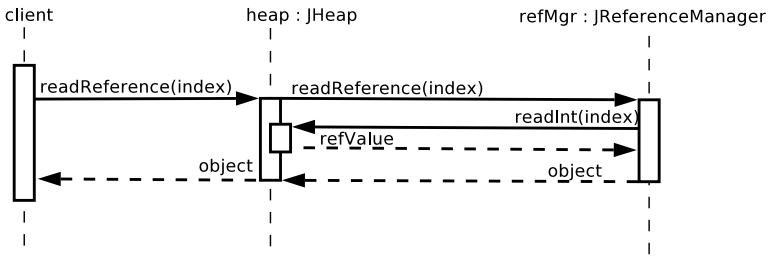

FIG. 6.17 – Implantation par défaut de la lecture d'une référence dans le *tas*.

Le composant *fabrique de gestionnaires de disposition* `JLayoutFactory`**.** Pour déterminer où lire et écrire au sein de la mémoire qui leur est attribuée, chaque objet `JClassInstance` et `JArray` utilise un *gestionnaire de disposition*, qui est un élément interne de type abstrait `JLayout` créé par le composant *fabrique abstraite de gestionnaires de disposition*. Dans le cas des instances de classes, le type du *gestionnaire de disposition* est `JClassLayout`. En fonction de la description d'un champ, il permet de localiser le premier octet dans la mémoire correspondant à la valeur de celui-ci. De même, dans le cas des tableaux, le type du gestionnaire de contenu est `JArrayLayout`. Il permet de déterminer où lire et écrire la valeur d'une entrée du tableau. Le composant *tas* ne créé qu'un seul gestionnaire de disposition par type d'objet à créer.

Dans le cas d'un *gestionnaire de disposition* d'une instance de classe de type `JClassLayout`, l'opération principale décrite ci-après, est fonction de la description d'une variable d'instance :

- `int getPosition(JField field)` : cette méthode retourne la position relative de la variable d'instance décrite par `field`, au sein de la mémoire attribuée à une instance de la classe associée à ce gestionnaire.

Les *gestionnaire de disposition* des tableaux possèdent tous la même opération `getPosition`, qui est fonction de l'indice d'une entrée :

- `int getPosition(int index)` : cette méthode retourne la position relative d'une entrée d'indice spécifiée, au sein de la mémoire attribuée à un tableau dont le type des éléments est associé à ce gestionnaire.

Dans chaque cas, le calcul de la position relative d'une variable d'instance ou d'une entrée d'un tableau est effectué en fonction de la taille des types primitifs décrit par le composant *tas*.

Le composant *allocateur* `JHeapAllocator`**.** Il possède la responsabilité de la création des instances de classes et des tableaux qu'il représente respectivement par les éléments internes de type `JClassInstance` et `JArray` (figure 6.18.

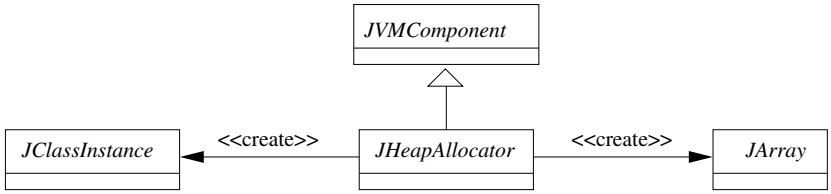

FIG. 6.18 – Le composant *allocateur*.

Il attribue à chacun de ces objets une zone de mémoire au sein du tas en calculant leur position *absolue*, le gestionnaire de disposition correspondant à la classe de l'objet ou celui des éléments du tableau et, leur assigne une référence *via* le service `assignReference` du composant *gestionnaire des références*.

Le service de création des instances de classes, `allocateJClassInstance` est par exemple illustré à la figure 6.19. Le composant *allocator* commence par attribuer un nouveau gestionnaire de disposition au nouvel objet `JClassInstance` puis, *via* le composant *gestionnaire des associations*, lui attribut une référence externe.

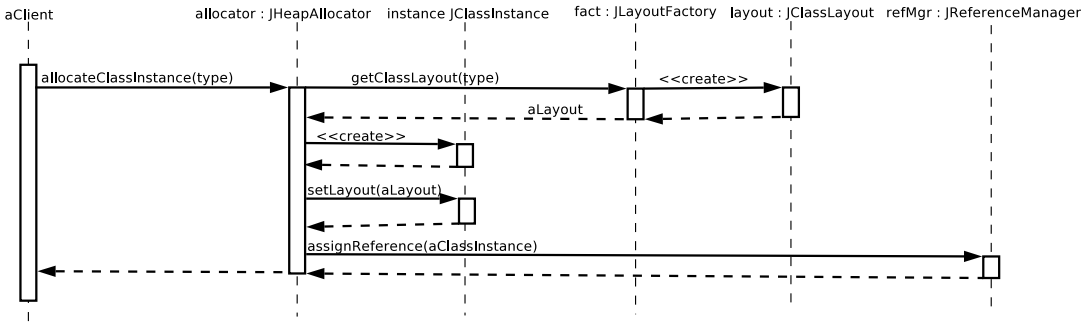

FIG. 6.19 – Allocation d'une nouvelle instance de classe.

6.4.2.2 Le moteur d'exécution

Le moteur d'exécution de la machine virtuelle Java est représenté par l'ensemble des composants suivant :

- le chargeur de classes et le moteur de résolution dynamique,
- des fils d'exécutions,
- un ordonnanceur, chargé de la gestion de ces fils d'exécution,
- une pile découpée en *frame* pour chaque fil d'exécution.

Le composant *chargeur de classes* `JClassLoader`**.** Corosol ne possède qu'un unique chargeur de classes représenté par le composant de type abstrait `JClassLoader`. Dans l'architecture d'une machine virtuelle Java standard, il joue le rôle du chargeur de classes de *bootstrap* et est responsable de la création des éléments de la *zone de données* (voir chapitre 4) définie par la spécification de la machine virtuelle Java [LY99]. Ces éléments sont de trois sortes :

- les informations sur les classes, avec en particulier la table `constant_pool` (voir chapitre 3) ;
- les informations sur les méthodes, avec en particulier les attributs de type `Code` (voir chapitre 3), qui contiennent les instructions de *bytecode* ;
- les informations sur les variables d'instances et de classes.

Le composant *chargeur de classes* créé trois types d'éléments internes : `JClass`, `JMethod` et `JField`, qui décrivent lors de l'exécution les informations relatives aux classes, à leurs méthodes et à leurs champs, respectivement. Chaque table `constant_pool` est également représenté par un élément interne de type `JConstantPool` accessible depuis chaque élément interne `JClass`. Les éléments internes `JClass` sont les méta-classes de Corosol. Elles possèdent des méthodes qui permettent de manipuler la pile des opérandes

d'une *frame*. Le chapitre suivant montrera comment elles sont effectivement utilisées.

En plus de la création des éléments de la *zone de données* lors du chargement du code compilé d'une classe, le composant *chargeur de classes* est responsable de la *résolution dynamique* utilisée pour transformer les références symboliques du *constant pool* en références concrètes. Autrement dit, il est responsable de la transformation des chaînes de caractères contenues par les tables `constant_pool` en objets `JClass`, `JMethod` et `JField`. Cette opération de résolution se décline en quatre services du composant `JClassLoader` :

- `resolveClass`, qui effectue la résolution d'une classe à partir d'une chaîne de caractères représentant son nom ;
- `resolveInterfaceMethod`, qui effectue la résolution d'une méthode d'interface à partir des chaînes de caractères représentants son nom, sa signature et le nom de l'interface qui la déclare ;
- `resolveMethod`, qui effectue la résolution d'une méthode qui n'est pas une méthode d'interface, à partir des mêmes éléments utilisés par le service `resolveInterfaceMethod` ;
- `resolveField`, qui effectue la résolution d'une variable d'instance ou de classe, à partir des chaînes caractères correspondant à son nom et au nom de la classe qui la déclare.

L'implantation par défaut de ces quatres services sollicitent les composants *recherche de méthodes* et *recherche de champs* que nous décrivons maintenant.

Les composants *recherche de méthodes* `JMethodLookup`**.** Le composant *recherche de méthodes*, de type abstrait `JMethodLookup` effectue la recherche d'une méthode parmi les super-classes/ou et les super-interfaces d'une classe donnée. Ces composants travaillent en collaboration avec le composant *chargeur de classes* lors de la de résolution dynamique de champs ou de méthodes.

La figure 6.20 illustre comment est réalisée la résolution dynamique d'une méthode. Elle est tout d'abord initiée par le service `resolveMethod`. Ensuite, conformément à l'algorithme décrit dans la spécification de Sun, la classe déclarant la méthode est chargée, puis cette méthode est recherchée parmi ses méthodes, les méthodes de ses super-classes, voire de ses super-interfaces, si les recherches précédentes n'aboutissent pas. Chaque recherche est représentée par un appel aux services suivants du composant `JMethodLookup` :

- `lookup(JClass c, String name, String descriptor)`, qui effectue la recherche de la méthode de nom `name` et de signature `descriptor`

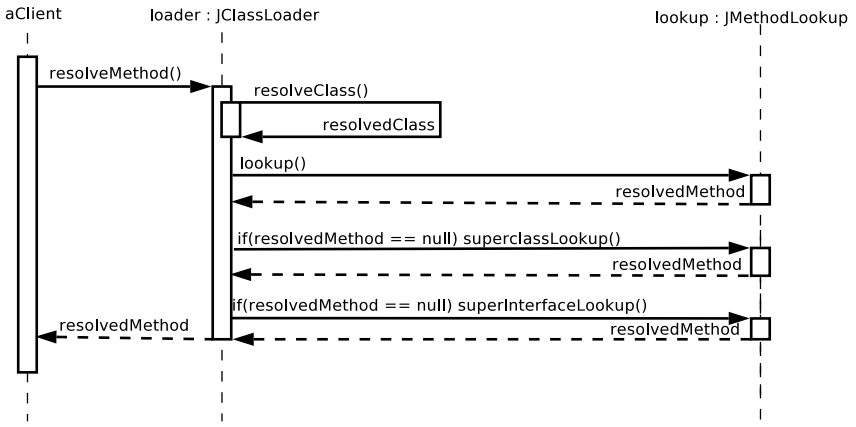

FIG. 6.20 – Résolution d'une méthode avec les composants `JClassLoader` et `JMethodLookup`.

au sein de la classe représentée par l'élément interne `c`, puis la retourne sous la forme d'un élément interne de type `JMethod` ;

- `superclassLookup(JClass c, String name, String descriptor)`, qui effectue la recherche de la méthode de nom `name` et de signature `descriptor` au sein des super-classes de la classe représentée par l'élément interne `c`, puis la retourne sous la forme d'un élément interne de type `JMethod` ;

- `superInterfaceLookup(JClass c, String name, String descriptor)`, qui effectue la recherche de la méthode de nom `name` et de signature `descriptor` au sein des super-interfaces de la classe représentée par l'élément interne `c`, puis la retourne sous la forme d'un élément interne de type `JMethod`.

Les composants *recherche de champs* `JFieldLookup`**.** Le composant *recherche de champs*, de type abstrait `JFieldLookup` effectue la recherche d'une variable d'instance ou de classe parmi les super-classes/ou et les super-interfaces d'une classe donnée. Ces composants travaillent en collaboration avec le composant *chargeur de classes* lors de la de résolution dynamique de champs ou de méthodes.

Des services similaires au composant `JMethodLookup` sont contenues au

sein du composant *recherche de champs*. Seul leur type de retour est différent ; il s'agit alors d'un élément interne de type `JField` qui représente le champ d'une classe.

Le composant *ordonnanceur* **:** `JScheduler`**.** L'ordonnanceur de la machine virtuelle Java est représenté dans Corosol le composant de type abstrait `JScheduler`. Il est le gestionnaire des *fils d'exécution* (*thread*) de notre machine virtuelle, chacun étant un élément interne de type abstrait `JThread`. Le composant `JScheduler` est chargé d'en assurer une exécution équitable, en fonction de leur priorité respective, selon l'algorithme d'ordonnancement qu'il implante.

Le modèle du composant *ordonnanceur* de Corosol ne prend pas encore en compte les moniteurs et les caches mémoires des différents fils d'exécution. Il se veut très simple. L'implantation par défaut du composant *ordonnanceur* de Corosol possède ainsi l'algorithme d'ordonnancement suivant :

Soit `queue`, la file des fils d'exécution, triés selon leur ordre de création.
Tant que la file `queue` n'est pas vide **faire**
  Extraire le fil d'exécution en tête de la file `queue` ;
  Soit `n`, la valeur de la priorité de ce fil d'exécution,
    Exécuter `n` instructions au sein de celui-ci ;
  Si toutes les instructions n'ont pas été exécutées **alors**
    Ajouter ce fil d'exécution à la fin de la file `queue`.
  FinSi
FinTantQue

Les services du composant *ordonnanceur* sont les suivants :

- `getCurrentThread()` qui retourne le fil d'exécution (processus léger) couran, retourné sous la forme d'un objet de type `JThread` ;
- `enqueueThread` qui ajoute un nouveau fil d'exécution dans la file des processus légers de l'ordonnanceur ; lui aussi est de type `JThread` ;
- `schedule()` qui initie la boucle d'ordonnancement de Corosol ; cette boucle peut être interrompue `via` la méthode suivante ;
- `breakScheduling()` qui interrompt la boucle d'ordonnancement démarrée avec la méthode `schedule()` ; du fait de l'atomicité des instructions dans notre architecture, cette interruption ne sera effective qu'après l'exécution de l'instruction du fil d'exécution courant.
- `getThreads()` qui retourne la liste des fils d'exécution non terminés
- `hasNext()` qui indique s'il existe un fil d'exécution dans la liste de l'or-

donnanceur.

La figure 6.21 illustre l'ordonnancement de deux fils d'exécution. Celui-ci est initié par la méthode `schedule` du composant *ordonnanceur* selon l'algorithme précédent. Le composant `Scheduler` demande à chaque fil d'exécution d'interpréter un certain nombre d'instructions *via* des appels à la méthode `void execNextInstruction(int n)` du type abstrait `JThread`.

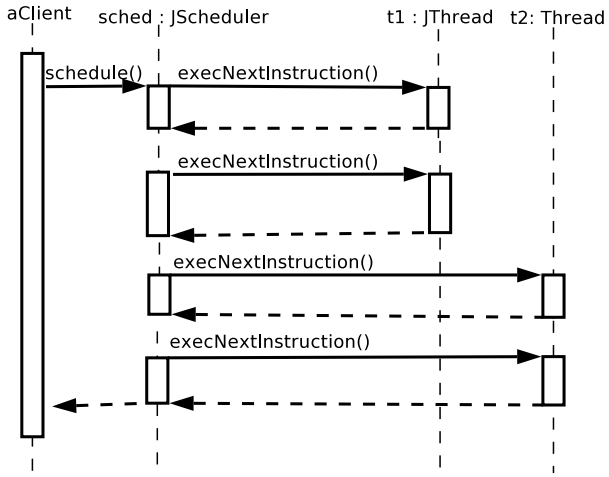

FIG. 6.21 – Le composant *ordonnanceur* et les fils d'exécution.

Examinons maintenant les composants de Corosol qui modélisent les instructions de la machine virtuelle.

Les composants *instructions* `JInstruction`**.** Les composants *instructions* représentent les instructions à exécuter par un fil d'exécution `JThread`. L'intérêt de modéliser la machine virtuelle Java jusque dans ses instructions est d'obtenir une architecture à *grain fin*, mais aussi de pouvoir redéfinir simplement par la suite le comportement de chacune d'entre elles, grâce à une localisation précise de leur sémantique.

Pour chacune des instructions de *bytecode* de la machine virtuelle, il existe un composant *instruction*, et chacun possède son propre type abstrait. Cependant, pour qu'un fil d'exécution les manipule de manière uniforme, tous ont le du même super-type, `JInstruction`. Son service `void exec(JThread thread)` est en charge de la réalisation de l'instruction. La figure 6.22 montre par exemple le diagramme de classes de quelques composants *instructions*, à

savoir `JInstruction.IAdd` qui est le type abstrait représentant l'instruction de *bytecode* `iadd`, `JInstruction.GetField` et `JInstruction.LLoad`, représentant respectivement `getfield` et `lload`.

Un composant de type abstrait `JInstruction` dépend statiquement de l'élément interne *fil d'exécution* de type `JThread`. Ce point est à souligné, car les instructions peuvent en modifier l'état, comme c'est le cas par exemple avec les instructions de branchement conditionnel.

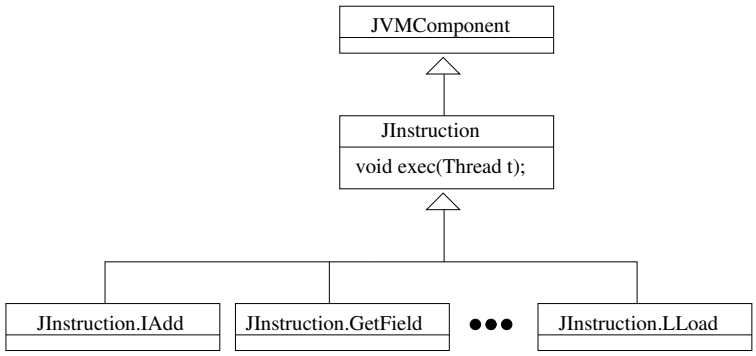

FIG. 6.22 – Les composants *instructions*.

Les éléments internes *frames* `JStackFrame`**.** Une nouvelle *frame* est allouée sur la pile d'un fil d'exécution à chaque fois qu'une méthode doit s'exécuter (voir chapitre 4). Elles sont modélisées par les éléments internes `JStackFrame`.

La spécification de la machine virtuelle met en évidence l'incidence de l'exécution d'une instruction sur l'état de la *frame* d'une méthode. L'instruction `imul`, par exemple, effectue le dépilement de deux valeurs de type `int` de la pile des opérandes puis y empile le résultat de leur produit. C'est la raison pour laquelle un fil d'exécution `JThread` possède la méthode `getCurrentFrame` qui retourne la *frame* courante (voir chapitre 4) sous la forme d'un élément interne de type `JStackFrame`, qui possède les méthodes suivantes (*Type* est primitif) :

- `void push`*Type*`(`*Type* `value)` : elle empile selon son type la valeur spécifiée sur la pile des opérandes de la *frame* ; un appel à la méthode `pushFloat(4.5)` empilera par exemple la valeur `4.5` ;

- *Type* `pop`*Type*`()` : elle retourne la valeur située au sommet de la pile des opérandes de la *frame* après l'avoir dépilée selon son type ; `popLong()` y dépilera par exemple une valeur de type `long` ;

- *Type* `load`*Type*`(int index)` : elle retourne, selon son type, la valeur lue à la position spécifiée dans le tableau des variables locales de la *frame* ; un appel à `loadInt(4)`, par exemple, donnera la valeur située à la position d'indice 4 de ce tableau ;

- `void store`*Type*`(`*Type*`value, int index)` : elle stocke, selon son type, la valeur spécifiée à la `index`-ième position du tableau des variables locales de la frame ; `storeDouble(7.9741, 0)` permettra par exemple d'y stocker la valeur `7.9741` à la position `0`.

À chaque instruction de *bytecode* est associée un type concret de composant implantant le comportement de celle-ci. La figure 6.23 nous montre par exemple le comportement de `IAdd` qui est le type concret du composant `JInstruction.IAdd` représentant l'instruction de *bytecode* `iadd`. Celle-ci dépile deux entiers de la pile des opérandes de la *frame* courante, pour ensuite y empiler la valeur de l'addition de ces deux valeurs. Elle nous montre également la collaboration entre un composant `JInstruction` et un élément interne `JStackFrame` qui represente une *frame* allouée pour l'exécution d'une méthode : le composant `JInstruction` utilise ici les méthodes `popInt` et `pushInt` du type abstrait `JStackFrame` pour respectivement dépiler et empiler des valeurs entières de type `int`.

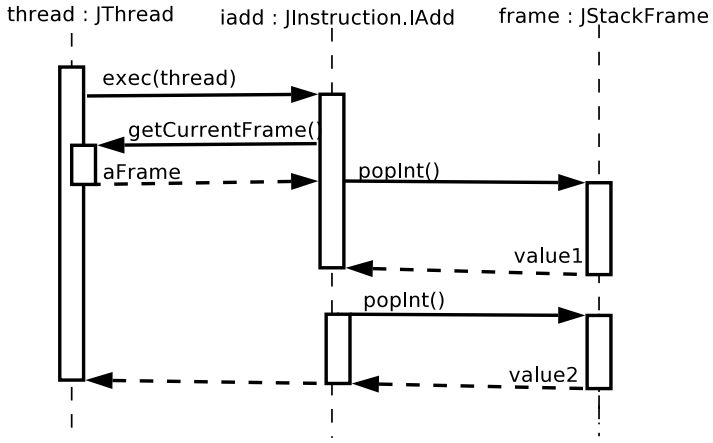

FIG. 6.23 – Exemple de l'exécution de l'instruction `iadd`

En plus de modifier la `frame` courante, une instruction peut avoir une incidence sur le fil d'exécution dans lequel elle s'exécute. C'est la raison pour

laquelle il est le paramètre principale de la méthode `exec` du type abstrait `JInstruction` afin de faire collaborer les composants *instructions* et les fils d'exécution. Il est utilisé en particulier par les implantations des instructions de branchements. La figure 6.24 montre comment l'instruction `goto` de la machine virtuelle a été réalisée. Son objectif est de modifier le compteur ordinal d'un fil d'exécution, ce qui est effectué par un appel à la méthode `incPC` du type abstrait `JThread`.

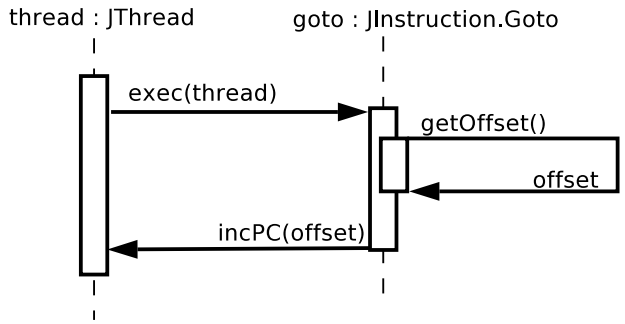

FIG. 6.24 – Exemple de l'exécution de l'instruction `goto`

Parmi les instructions de *bytecode*, on distingue celles qui provoque l'invocation de méthodes. Les composants associés dépendent dynamiquement du composant *invocation de méthodes* que nous décrivons ci-après.

Le composant *invocation de méthodes* `JMethodInvoker`**.** Le schéma d'exécution normal de Corosol est d'interpréter chacune des instructions de *bytecode* transformées en composants *instructions*. Cependant comme le montre le chapitre suivant, il est parfois nécessaire de déléguer l'exécution à la machine virtuelle *hôte*. C'est en particulier le cas lors l'exécution d'un service d'un composant depuis l'application, d'une méthode est native ou simplement si le programmeur de Corosol l'a décidé. Ce mécanisme de délégation de l'exécution est fondamental dans l'architecture Corosol. Le composant *invocation de méthodes* possède cette responsabilité, à travers les deux méthodes suivantes :

- `void invokeMethod(JMethod method, JThread thread)` : si la méthode décrite par `method` est native ou reflète un service d'un composant, `invokeMethod` délègue l'exécution à la machine virtuelle *hôte*, puis elle empile le résultat éventuel sur le sommet de la pile des opérandes de la `frame` courante du fil d'exécution désignée par `thread` ; sinon elle prépare l'exécution de `method` au sein du fil d'exécution `thread` en sauvant le contexte d'exécution courant ;

– `void invokeConstructor(JConstructor cons, JThread thread)` : si le constructeur décrit par `cons` doit créer un objet qui ne doit pas être alloué physiquement au sein du tas de Corosol, `invokeConstructor` l'exécute au sein de la machine virtuelle *hôte* et empile la référence du nouvel objet sur la pile des opérandes de la frame courante du fil d'exécution désigné par `thread` (le chapitre suivant montrera que cette référence n'est pas la «vraie» référence de l'objet au sein du tas de la machine virtuelle *hôte*, mais plutôt une référence qui lui est attribuée au sein du tas de Corosol) ; sinon elle prépare l'exécution de ce constructeur au sein du fil d'exécution `thread` en sauvant le contexte d'exécution courant.

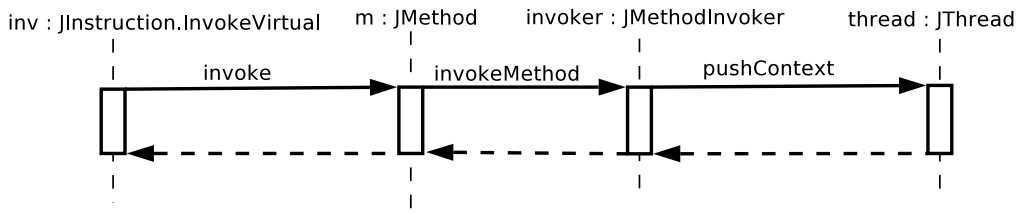

FIG. 6.25 – Le composant *invocation de méthode*

Le composant *invocation de méthode* est sollicité par le biais des composants *instructions* qui représentent les instructions de *bytecode* initiant l'exécution d'une méthode, à savoir :

– l'instruction `invokestatic`, représentée par le composant de type abstrait `JInstruction.InvokeStatic` et responsable de l'exécution d'une méthode statique ;
– l'instruction `invokeinterface`, représentée par le composant de type abstrait `JInstruction.InvokeInterface` et responsable de l'exécution d'une méthode d'interface ;
– l'instruction `invokevirtual`, représentée par le composant de type abstrait `JInstruction.InvokeVirtual` et responsable de l'exécution d'une méthode qui n'est ni une méthode d'interface, ni un constructeur, ni une méthode statique et ni une méthode privée ;
– l'instruction `invokespecial`, représentée par le composant de type abstrait `JInstruction.InvokeSpecial` et responsable de l'exécution d'une méthode privée ou d'un constructeur.

La figure 6.25 montre comment le composant *invocation de méthode* est utilisé et comment il collabore avec un fil d'exécution, lors de l'invocation d'une méthode *via* l'instruction de *bytecode* `invokevirtual`. Elle décrit le schéma d'exécution normal de Corosol, c'est-à-dire sans délagation de l'exécution de cette méhode à la machine virtuelle *hôte*. Dans le cas présenté ici, le contexte d'exécution est sauvegardé *via* l'opération `pushContext` du type `JThread`.

6.5 Conclusion

Dans ce chapitre, nous avons explicité l'architecture de notre machine virtuelle Java, Corosol. Dans celle-ci, nous considérons la machine virtuelle Java comme un conteneur de composants. Chacun d'entre eux peut être consulté et remplacé par l'intermédiaire de l'application qui est exécutée. Cela est mise en œuvre par des mécanismes de réflexivité que nous décrivons dans le chapitre suivant.

Chapitre 7

Réflexivité dans Corosol

7.1 Introduction

Les mécanismes de réflexivité de Corosol permettent à un programme de consulter son propre état (comme ses classes ou ses méthodes, par exemple) et de le modifier. L'application peut également consulter et modifier les composants de Corosol : elle peut ainsi agir sur la sémantique de sa propre exécution. Ce chapitre définit Corosol comme un interprète réflexif et explique la réalisation des mécanismes sous-jacents.

7.2 Modèle d'exécution de Corosol

Comme le chapitre 6 l'a montré, Corosol est une machine virtuelle Java entièrement écrite en Java. Elle est ainsi exécutée par une *seconde* machine virtuelle Java, que nous avons désignée par *hôte* (figure 7.1).

Dans ce modèle d'exécution, Corosol représente l'application sous la forme d'objets Java, tandis que ses composants et ses éléments internes sont représentés sous une forme native par la machine virtuelle hôte. Il peut s'agir, par exemple du langage C, si HotSpot [Hot], la machine virtuelle Java de Sun Microsystems, est utilisée comme *hôte*. Nous pouvons tirer de ce modèle d'exécution des propriétés réflexives pour Corosol.

7.2.1 Méta-niveaux d'exécution

La machine virtuelle *hôte*, Corosol et l'application forment trois couches distinctes dans la modélisation de l'exécution de l'application, les deux premières étant elles-mêmes des plate-formes d'exécution. La raison essentielle

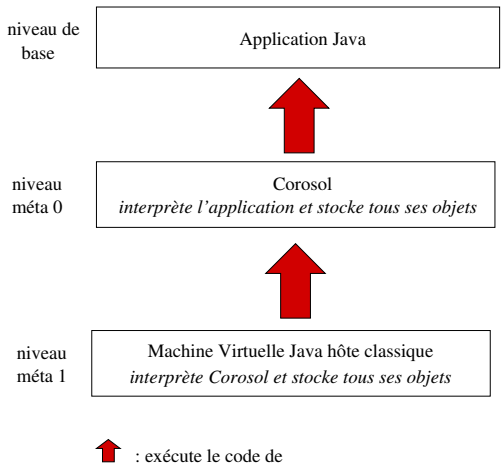

FIG. 7.1 – Modèle d'exécution de Corosol.

de la superposition des deux premières couches est l'enrichissement de l'interface d'introspection pour pouvoir permettre à l'application de modifier son interprète, l'interprète transformé étant représenté par Corosol. Dans ses travaux sur 3-Lisp, Brian C. Smith [Smi82, Smi84] généralise cette idée : la modification par l'application de son interprète i_k donne naissance à un nouvel interprète i_{k+1} qui, dans un modèle en couches, vient se placer entre l'application et i_k. Le nouvel interprète i_{k+1} exécute l'application, et est lui même exécuté par l'ancien interprète (non modifié) i_k. Cette modélisation produit un modèle qui possède un nombre fini, mais non borné, de couches à un instant t. Dans le cas de Smalltalk, les objets qui composent l'interprète sont des instances de classes, elles-mêmes instances de méta-classes. Il existe ainsi une régression infinie du lien d'instanciation. Cependant, elle n'est qu'apparente et est court-circuitée par le fait que chaque méta-classe est une instance d'une méta-classe unique (`MetaClass`), gérée par l'interprète. Le nombre de méta-niveaux d'exécution est donc réduit à deux : l'application et l'interprète.

Notre modèle d'exécution définit les trois niveaux d'exécution suivants :
- le *niveau de base*, qui représente l'application Java à exécuter,
- le *niveau méta 0*, qui est Corosol,
- le *niveau méta 1*, qui est la *machine virtuelle hôte* ; ce dernier niveau est une machine virtuelle Java *native*, c'est-à-dire non interprétée par une autre machine virtuelle Java mais exécutée par un système d'exploitation.

Comme dans 3-Lisp [Smi82, Smi84], à l'exception du niveau de base, chaque niveau constitue l'interprète du niveau qui lui est immédiatement supérieur (voir figure 7.2).

Appartenance à un méta niveau. Un objet appartient au niveau méta 0 s'il représente un objet du niveau de base et s'il possède une référence externe dans le tas de Corosol. Il est physiquement alloué dans le tas de Corosol.

Un objet appartient au niveau méta 1 s'il représente un objet du niveau méta 0 et s'il possède une référence dans le tas de la machine virtuelle hôte. Il est physiquement alloué dans le tas de la machine virtuelle hôte.

Le découpage de l'exécution en méta-niveaux permet, dans ce qui suit, de définir Corosol comme un interprète réflexif.

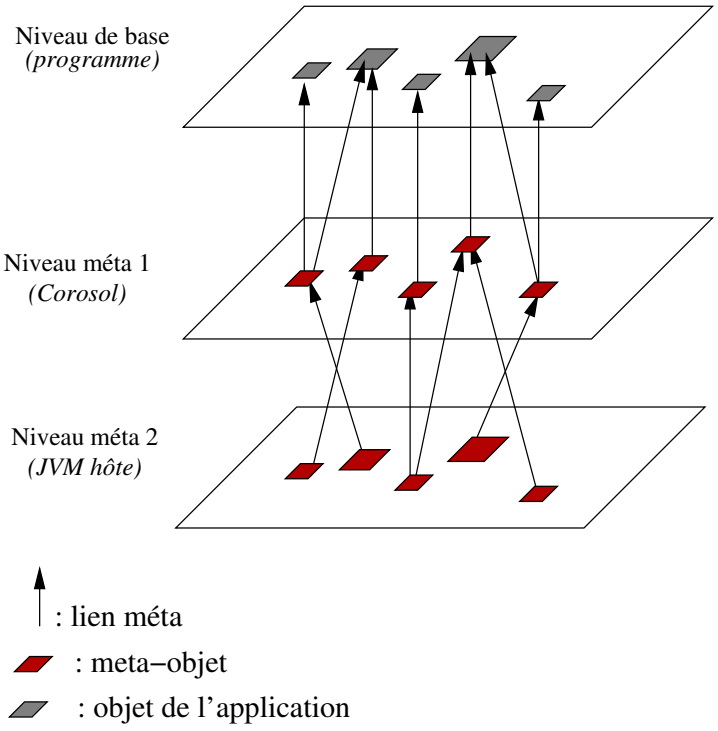

FIG. 7.2 – Les méta-niveaux dans Corosol.

7.2.2 Un interprète réflexif

Le chapitre 2 a montré ce qui caractérise un interprète réflexif. Il rend accessible au niveau de base, c'est-à-dire à l'application, des données de son niveau, le niveau méta, qui représentent son état d'exécution. Dans Corosol, le niveau de base et le niveau méta sont liés causalement : toutes modifications de l'état d'exécution depuis le niveau de base impliquent des modifications de cet interprète au niveau méta. Son modèle d'exécution permet à Corosol d'être dans cette catégorie d'interprète.

7.2.2.1 Réflexivité *totale*

Tous les langages ne permettent pas une expression complète de l'état d'exécution de leur interprète. Certains ne proposent que des possibilités d'introspection (*ie.* de consultations) et/ou des possibilités d'intercessions (*ie.* de modifications) limitées. En Java par exemple, le protocole à méta-objets fourni par le paquetage `java.lang.reflect`, permet d'examiner les classes de l'application, ses méthodes et ses champs. Le code source, ci-après, illustre son utilisation, en montrant comment la liste des méthodes de la classe d'un objet de l'application peut être affichée :

```
import java.lang.reflect.*;
import java.awt.*;

public class Example {
  public static void main(String[] args) {
    Polygon p = new Polygon();
    Class c = p.getClass();
    Method[] methods = c.getMethods();
    for (int i = 0; i < methods.length; i++) {
      System.out.println(methods[i]);
    }
  }
}
```

Java permet ainsi l'introspection partielle du programme. Par exemple, la granularité du protocole à méta-objets ne permet pas d'accéder au code d'une méthode. Il ne permet pas non plus la modification des éléments de l'implantation de sa machine virtuelle depuis l'application, comme par exemple l'algorithme de ramasse-miettes ou la représentation des objets en mémoire. Le protocole à méta-objets qu'il fournit ne permet pas une réflexivité *totale* [MJD96]. En revanche, Corosol dispose de cette propriété de réflexivité totale.

En effet, en utilisant son protocole à méta-objets, une application peut, à tout moment, en examiner et en changer les composants. Elle peut ainsi modifier la sémantique de sa propre exécution. De plus, à l'image de celui fourni *via* paquetage `java.lang.reflect`, l'application peut également accéder aux instances des méta-classes correspondant à ses classes, à leurs champs et à leurs méthodes. Chaque élément de l'implantation de Corosol est ainsi rendu disponible à l'application ou, pour reprendre notre terminologie, chaque élément du niveau méta 1 peut être examiné et/ou remplacé depuis le niveau de base. Son protocole à méta-objets est de plus qualifié d'*implicite* [Zim96], car les composants et les éléments internes rendus visibles depuis le niveau de base participent de façon transparente à l'implantation de l'exécution du niveau méta 0. À l'opposé, celui du paquetage `java.lang.reflect` est *explicite* car les instances des méta-classes `Method`, `Field` (voire `Proxy`) ne sont utilisées qu'explicitement par l'application. Ils ne participent aucunement à l'implantation de l'exécution de l'application, tout du moins, pas systématiquement. Par exemple, la méthode `invoke` de la classe `Method` n'est pas nécessairement utilisée implicitement par une implantation de la machine virtuelle Java pour réaliser chacun des appels de méthodes survenant lors de l'exécution. Ces objets ne sont ainsi réifiés que pour satisfaire les appels réflexifs de l'application.

Corosol est donc un interprète réflexif *total*. Le programme ci-après illustre son protocole à méta-objets. Il montre comment les composants sont accessibles depuis l'application.

```
1  public class HelloWorld {
2    public static void main(String[] args) {
3      System.out.println("Hello world! (standard)");
4      JVirtualMachine jvm = Corosol.getVirtualMachine();
5      jvm.replaceComponent(new JMMFInvokeVirtual());
6      System.out.println("Hello world! (JMMF-ly)");
7    }
8  }
```

À la ligne 4, l'application obtient un objet de type `JVirtualMachine` qui représente le composant *conteneur de composants* de Corosol. À partir de celui-ci, elle peut modifier la sémantique de son exécution : la ligne 5 montre, par exemple, le remplacement du composant réalisant le comportement de l'instruction de *bytecode* `invokevirtual` par le composant `JMMFInvokeVirtual`.

Ce programme vient d'illustrer le protocole à objet implicite de Corosol, mais aussi que ses composants sont utilisés par l'application comme n'importe

quel autre objet Java, à l'image des chaînes de caractères, par exemple. Cette propriété importante s'appelle *symbiose linguistique*.

7.2.3 Symbiose linguistique

Dans notre modèle d'exécution, le programme est évalué, manipulé, et surtout *représenté* dans le *même* langage que son interprète, c'est-à-dire Java. Il peut modifier la sémantique de sa propre exécution *via* le protocole à méta-objets de Corosol, dans des opérations exprimées, *mais aussi implantées*, en Java.

Il s'agit alors de faire raisonner (c'est-à-dire examiner et/ou modifier) le niveau méta 0 *via* le niveau de base. En d'autres termes, en désignant par *langage de base* le langage dans lequel l'application est écrite et par *langage méta* celui dans lequel est écrit son interprète, il s'agit alors de faire raisonner le langage méta par l'intermédiaire du langage de base. Dans de nombreux cas, ces deux langages sont différents. C'est le cas par exemple avec SOUL [BGW02, WD01], qui est un langage de logique implanté en Smalltalk et capable de raisonner sur des objets SOUL et Smalltalk. C'est encore le cas avec Rbcl [IMY92] qui est un langage implanté en C++ capable de raisonner avec celui-ci.

7.2.3.1 Définition

Un langage donné est dit en *symbiose linguistique avec son langage d'implantation* [IMY92, GWDD06] s'il possède la capacité d'exprimer et de manipuler les objets du langage méta avec le langage de base.

Dans le cas de Corosol, l'application peut manipuler ses composants comme n'importe quel autre objet Java. Ainsi, lorsqu'il interprète ce programme, Corosol doit faire face à deux types d'objets :

- la représentation en Java des objets de l'application,
- la réification de ses propres composants et éléments internes.

Comme nous l'aborderons plus loin dans ce chapitre, les composants ne sont qu'un cas particulier des objets dit *natifs*, alloués dans le tas de la machine virtuelle *hôte*. Il s'agit par exemple de tableaux ou encore des chaînes de caractères.

7.3 Implantation de la réflexivité

Après avoir examiné les propriétés réflexives du modèle d'exécution de Corosol, nous expliquons maintenant comment l'examen et l'utilisation des objets des méta-niveaux inférieurs sont réalisés depuis l'application.

7.3.1 Une manipulation uniforme des objets

Les propriétés réflexives de Corosol impliquent que les objets provenants de l'application puissent être manipulés comme des objets provenant de Corosol, et réciproquement. En d'autres termes, les objets doivent pouvoir être manipulés uniformément depuis l'application ou depuis Corosol, indépendamment de leur provenance. C'est en fonction de celle-ci que notre machine virtuelle détermine comment exécuter les méthodes d'un objet :

- s'il provient du niveau de base, alors la pile de Corosol est utilisée, comme expliqué au chapitre 6,
- dans le cas contraire, l'exécution est déléguée à la machine virtuelle *hôte*.

Pour limiter au maximum les vérifications de la provenance de chaque objet avant de pouvoir exécuter une de ses méthodes, Corosol pourvoit à une représentation commune de chaque objet qu'il manipule.

7.3.1.1 Un super-type commun : `JHeapObject`

Le chapitre précédent a montré comment les objets du niveau de base sont représentés. Ainsi, le type `JClassInstance` est utilisé pour représenter chaque instance de classe de l'application et `JArray` chaque tableau. Ils possèdent `JHeapObject` comme super-type commun qui sera également utilisé dans la représentation des objets ne provenant pas du niveau de base, comme les composants.

Selon la terminologie utilisée dans ce chapitre, un objet `JHeapObject` est la représentation en Java d'un objet du niveau de base. Il appartient au niveau méta 0 et est alloué physiquement au niveau inférieur, c'est-à-dire le niveau méta 1, qui est celui de la machine virtuelle *hôte*.

`JHeapObject` correspond à une partie de l'implantation de Corosol, c'est-à-dire un *élément interne* : il hérite donc du type `JObject` (*cf.* chapitre 6) qui représente de tels objets. Sa méthode `isNative` permet de déterminer si l'objet qu'il représente est du niveau de base ou est *natif*, c'est-à-dire provenant des méta-niveaux inférieurs.

7.3.1.2 Une gestion commune des références

Lors de l'interprétation de l'appel d'une méthode, la référence de l'objet appelant ainsi que la valeur de ses arguments sont stockés successivement dans le tableau des variables locales d'une nouvelle *frame* (voir chapitre 4). Puisque les objets de l'application sont représentés par d'autres objets Java du niveau méta 0, ces derniers possèdent une référence issue du niveau méta

1, celui de la machine virtuelle hôte. Ce sera également le cas des autres objets utilisés par cette application mais qui n'appartiennent pas au niveau de base.

Cependant, cette référence n'est pas directement manipulable en Java car il n'existe pas de type primitif associé aux références. En particulier, elle n'est pas recopiable dans une *frame* de Corosol : en effet, la représentation des références dans une *frame* de Corosol peut être différente de celle utilisée par la machine virtuelle *hôte*, et de plus, Corosol ne peut y accéder. C'est la raison pour laquelle Corosol utilise des références que nous avons qualifié d'*externes* au chapitre 6. Chacune est simplement une valeur numérique qui identifie chaque instance de classe et chaque tableau utilisés par l'application interprétée, quel que soit le méta-niveau d'origine. Corosol attribue donc à chaque objet `JHeapObject` une telle référence *via* le composant *gestionnaire des références* (chapitre 6).

La portion de programme suivante illustre l'attribution des références des objets de l'application. Un objet de type `Integer` (ligne 1), un objet de type `ArrayList` (ligne 2) et un tableau de caractères (ligne 3) y sont créés successivement.

```
1  Integer anInt = new Integer(5);
2  ArrayList list = new ArrayList();
3  char[] charArray = new char[]{'a', 'r', 'r', 'a', 'y'};
```

Le tableau 7.1 montre comment chaque objet créé dans ce programme, chacun issu du niveau de base, est représenté au niveau méta 0. Nous avons en plus fait apparaître les références externes attribuées pour chacun de ses objets.

Nom	**Type réel**	**Type du niveau méta 0**	**Référence** *externe*
`anInt`	Integer	JClassInstance	10
`list`	ArrayList	JClassInstance	11
`charArray`	char[]	JArray	12

TAB. 7.1 – Représentation des objets du niveau de base au niveau méta 0.

7.3.2 Réification des composants de Corosol

Les composants de Corosol doivent être réifiés pour être ensuite manipulés indifféremment par l'application comme n'importe quel de ses autres objets Java, afin de réaliser la symbiose linguistique entre le langage de base et le langage méta.

Le mécanisme que nous décrivons maintenant réalise cette opération de réification. Il possède des points communs avec le mécanisme *Up* décrit dans [Ste94, WD01] et que nous avons brièvement décrit au chapitre 2. Celui-ci a pour objectif la transformation de n'importe quelle entité (objet ou valeur d'un type primitif) allouée physiquement au niveau méta 1 en une entité du niveau méta 0 qui représente le niveau de base, c'est-à-dire un objet de type `JHeapObject`. Il nous permet en particulier de réaliser la réification des composants de Corosol, car il sont représentés par la machine hôte au niveau méta 1 et doivent, pour être réifiés depuis le niveau de base, posséder la bonne représentation au niveau méta 0.

Notre mécanisme *Up* se définit comme suit :
Si *value* est d'un type primitif **alors**
$Up(value) = wrapPrimitiveValue(value)$
Sinon Si *value* est la référence `null` **alors**
$Up(value) = wrapNullReference(value)$
Sinon Si *value* est un objet du niveau méta 0 **alors**
$Up(value) = value$
Sinon *value* désigne un objet du niveau méta 1 **et**
$Up(value) = wrapNativeObject(value)$

Si la valeur de l'argument *value* est une primitive, la référence `null` ou un objet du méta-niveau 1, l'opération *Up* encapsule *value* dans un objet de super-type `JHeapObject`. Cela est représenté dans l'utilisation des méthodes *wrapPrimitiveValue*, *wrapNullReference* et *wrapNativeObject*. Dans le cas où *value* représente un objet du niveau méta 0 représentant un objet de l'application, c'est-à-dire un objet de type `JClassInstance` dans le cas d'une instance de classe, ou `JArray` dans le cas d'un tableau, l'opération *Up* est l'identité : elle retourne simplement *value*, car celle-ci correspond déjà à un objet de super-type `JHeapObject`.

Pour que la réification d'un composant de Corosol soit complète, une référence *externe* est associée à l'objet issu de sa réification, c'est-à-dire un objet de type `JHeapObject` encapsulant effectivement ce composant, ce qui est illustré par la portion de programme suivante :

```
1  JVirtualMachine jvm = Corosol.getVirtualMachine();
2  JClassLoader loader = (JClassLoader)jvm.getVirtualMachine();
3  JClass c = loader.loadClass("java.util.Date");
```

Dans celle-ci, trois objets du méta niveau 0 sont utilisés, à savoir, le composant *machine virtuelle* qui est l'interprète du programme (ligne 1), le

composant *chargeur de classe* (ligne 2), et également un objet `JClass` correspondant à la classe d'un objet de type `java.util.Date` (ligne 3). Nous avons illustré dans le tableau 7.2 comment la gestion de ces objets est effectuée au niveau méta 0. Chacun des objets manipulés au niveau de base correspondant à un objet du niveau inférieur est encapsulé dans un objet sous-type de `JHeapObject` (ici ce sera un objet de type `NativeJClassInstance` (voir figure 7.3) et auquel une référence externe est attribuée.

Nom	**Type réel**	**Type encapsulant**	**Référence** *externe*
`jvm`	JVirtualMachine	NativeJClassInstance	47
`loader`	JClassLoader	NativeJClassInstance	23
`c`	JClass	NativeJClassInstance	78

TAB. 7.2 – Associations entre les références de méta-niveaux différents.

Comme nous venons de l'illustrer, même les composants de Corosol sont réifiés depuis l'application. Mais d'autres objets peuvent être aussi «visibles» (c'est-à-dire utilisés) depuis l'application. Ce sont des objets créés par la machine virtuelle hôte qui ne sont pas alloués dans le tas de Corosol. Par opposition aux objets de l'application, ces objets sont dit *natifs* car provenant du méta-niveau 1.

7.3.3 Réification des objets *natifs*

Il existe des instances de classes et de tableaux que Corosol n'alloue pas au sein de son tas, c'est-à-dire au niveau méta 0. Ces objets sont par exemple créés *via* des méthodes natives au sein du tas de la machine virtuelle hôte. Cependant, le programmeur (mais aussi l'application) peut décider de représenter certains objets du programme au niveau méta 0 et d'autres au niveau méta 1, c'est-à-dire directement dans le tas de la machine virtuelle hôte. Par

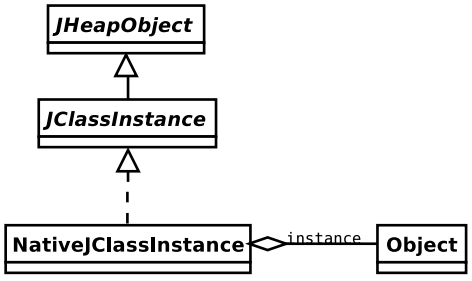

FIG. 7.3 – Réification des composants.

exemple, JavaInJava [Tai98], la machine virtuelle Java de Sun, entièrement écrite en Java, alloue *tous* les objets du programme directement dans le tas de sa machine virtuelle hôte. Une telle approche permet de résoudre le problème du changement de représentation des données entre les méta-niveax 1 et 0. Le mécanisme Up/Down devient alors inutile. Cependant il est impossible de changer leur représentation mémoire (contrairement à Corosol, voir section 10.5) car il s'agit de celle de la machine virtuelle *hôte* qui est native.

Nous appelons de tels objets *natifs*. Ils sont créés par la machine virtuelle *hôte*. Le programme à interpréter peut les manipuler directement. Pour rendre cela possible, et maintenir une évaluation uniforme des arguments de chaque méthode, les objets provenant du niveau méta 1 sont encapsulés dans des objets sous-type de `JHeapObject`. Ce procédé est comparable à l'encapsulation des composants dont nous avons précédemment parlé.

Par exemple, dans le cas d'un tableau alloué au niveau méta 1, nous utilisons la classe `NativeJArray` pour le convertir en un objet du niveau méta 0 (voir figure 7.4).

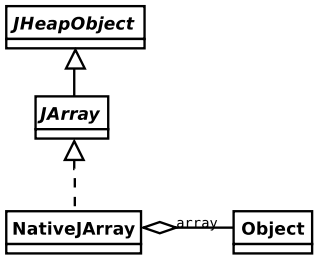

FIG. 7.4 – Les tableaux dits *natifs*.

7.3.4 Exécution des objets par l'application

Les composants de Corosol se voient attribuer une référence externe, au même titre que n'importe quelle instance de classe de l'application. Le mécanisme de résolution de méthodes pour un composant reste donc le même. Cependant deux types de méthodes peuvent être appelées depuis l'application :

- celles qui appartiennent à des objets de l'application,
- celles qui appartiennent à des composants de Corosol.

Seul change le mode d'exécution : le code de la méthode d'un composant (ou d'un objet natif) n'est pas interprété par Corosol. Il est exécuté par la machine virtuelle hôte par l'intermédiaire des mécanismes de réflexivité de celle-ci. Cependant, cette délégation de l'exécution nécessite une transformation des arguments de la méthode à exécuter. Le mécanisme utilisé est

comparable au mécanisme *Down* décrit dans [Ste94, WD01]. Celui-ci nous permet de transformer les objets du niveau méta 0 en des objets du niveau méta 1. Notre mécanisme *Down* se décrit comme suit :

Si *value* est la valeur d'un type primitif **alors**
$Down(value) = value$
Sinon si *value* est la valeur de la référence `null` **alors**
$Down(value) = null$
Sinon si *value* est un objet du niveau méta 1 **alors**
$Down(value) = value$
Sinon *value* est un objet du niveau méta 0 **alors**
$Down(value) = Proxy(value)$

Considérons une méthode `f` exécutée par Corosol. Supposons qu'elle effectue un appel d'une méthode `m` sur un composant de référence `ref` au sein du tas de Corosol. Comme `f` est exécutée par Corosol, cette dernière alloue une pile d'exécution pour cette méthode. Au niveau de Corosol, l'invocation de la méthode `m` a pour effet d'empiler `ref`, la référence du composant, sur la *frame* de la méthode `f`, ainsi que les différentes valeurs de ses arguments. Elle provoque aussi la résolution de la méthode `m`. Le résultat de cette résolution est un objet `JMethod` représentant la méthode à exécuter.

Puis, on détermine l'objet de la classe `java.lang.reflect.Method` équivalent à cet objet `JMethod` qui permettra à la machine virtuelle *hôte* d'exécuter la méthode du composant. Sa méthode `invoke` requiert en paramètre la référence de l'objet cible, c'est-à-dire l'objet sur lequel la méthode est appelée, ainsi qu'un tableau contenant la valeur des arguments. Tous ces paramètres sont situés sur la *frame* de la méthode appelant `m`, c'est-à-dire `f`. Avant de les transférer en tant qu'arguments de la méthode `invoke`, il faut les transformer selon le mécanisme `Down` que nous avons précédemment décrit.

Si le type de l'argument est primitif, le transfert de sa valeur se fait façon simple. La valeur de cet argument est dépilée de la pile de `f` en fonction de son type, puis est recopiée au sein du tableau d'arguments de la méthode `invoke` de l'objet `java.lang.reflect.Method`.

En revanche, si l'argument est une référence deux cas sont à considérer :

- si cette référence est celle d'un objet natif et alloué non au sein du tas de Corosol, mais au sein du tas de la machine virtuelle hôte (il peut être en particulier un composant), alors cette référence est dépilée de la pile de la méthode `f`, puis est recopiée au sein du tableau d'arguments de la méthode `invoke` ;
- sinon, cette référence est celle d'un objet alloué au sein du tas de Corosol. Or, au sein de Corosol, les instances de classes sont représentées

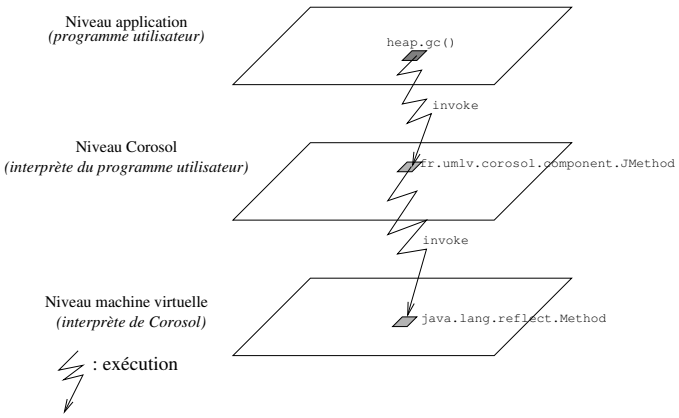

FIG. 7.5 – Appel de méthode sur un composant de Corosol.

par des objets de la classe `JClassInstance`. Un tel objet ne possède donc pas le même type dynamique de l'instance de classe qu'il représente. Cette référence est donc dépilée de la pile de la méthode `f` puis, l'objet `JClassInstance` associé à cette référence est encapsulé dans une autre instance de classe jouant le rôle de *proxy* et possédant cette fois le même type dynamique. Soit `C` la classe concrète représentant ce type dynamique. La classe du *proxy* `CProxy` est créée comme héritant de `C` et encapsulant, `instance` la référence de l'objet `JClassInstance` du tas de Corosol (le chapitre suivant décrira plus en détail cette création de *proxy*). Toutes les méthodes de la classe `C` sont surchargées dans `CProxy` : les accès en lecture et en écriture sont redirigés vers `instance`). Après création de la classe `CProxy`, la référence d'une instance de ce *proxy* est recopiée au sein du tableau d'arguments de la méthode `invoke`.

On retrouve la même problématique de passage d'arguments entre Corosol et la machine virtuelle hôte lors de l'appel de certaines méthodes natives. Ces méthodes natives sont celles qui font appel à des éléments externes à Corosol, comme par exemple, les méthodes d'entrées-sorties.

7.3.4.1 Application à la création des objets

L'application qui est exécutée par Corosol peut créer des objets qui sont des éléments internes ou des composants. Ils doivent être alloués physiquement au niveau méta 1, contrairement aux autres objets du programme qui le sont dans le tas de Corosol. Nous montrons maintenant comment le méca-

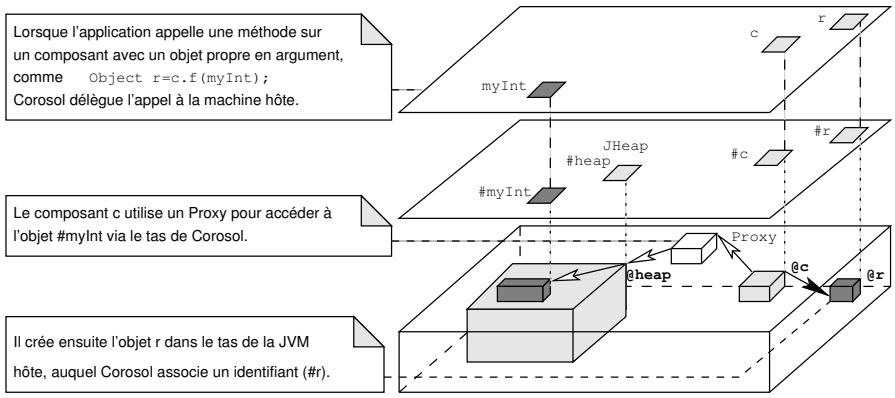

FIG. 7.6 – Allocation et références : composants de Corosol *versus* objets utilisateurs.

nisme *Down* s'applique dans ce cas.

`new` est l'instruction de *bytecode* initie la création d'une instance d'une classe. La référence de l'objet créé est ensuite empilée sur la *frame* courante.

Si la classe de cette nouvelle instance correspond à un objet l'application, elle n'est pas un sous-type de `JObject` et son instance doit être physiquement alloué dans le tas de Corosol. L'exécution de l'instruction `new` se déroule de façon standard (figure 7.7).

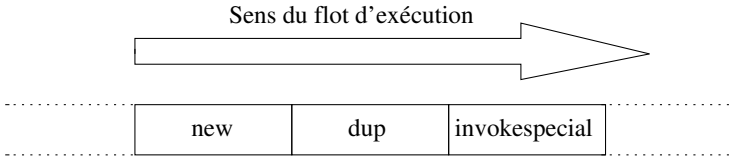

FIG. 7.7 – Flot d'exécution standard de l'instruction `new`.

Dans ce cas, le composant *instruction* de type `New`, responsable de l'implantation du comportement de l'instruction de *bytecode* du même nom, provoque l'allocation physique du nouvel objet dans le tas de Corosol, *via* le service `allocateClassInstance` du composant *allocateur* (dont le type abstrait est `JHeapAllocator`. La référence de cette nouvelle instance est ensuite empilée sur la *frame* courante *via* l'opération `pushReference` du type `JStackFrame` (figure 7.8).

Si la classe de l'instance à créer est être celle d'un composant (ou tout autre élément interne), elle est alors un sous-type de `JObject`, et l'objet concerné doit être physiquement alloué dans le tas de la machine virtuelle

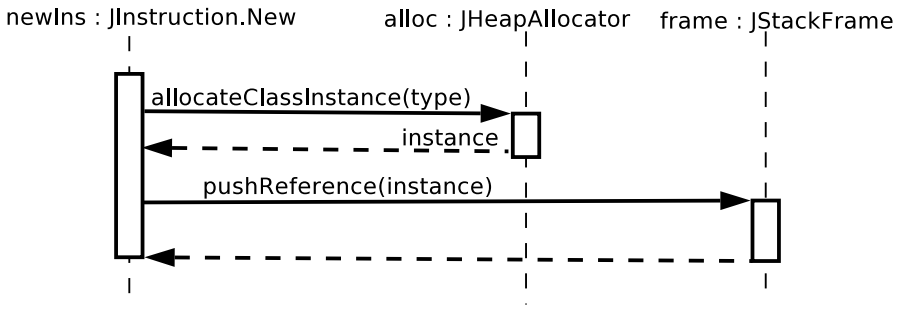

FIG. 7.8 – Allocation des objets de l'applications dans le tas de Corosol.

hôte. L'instruction `new` ne doit pas se dérouler de façon standard. Plutôt que d'empiler la référence externe d'une nouvelle instance, le comportement du composant `New` est d'empiler une référence fictive. La figure 7.9 montre le fil d'exécution associé : l'instruction de *bytecode* `dup` est normalement exécutée après chaque instruction `new`. Mais, dans ce cas présent, elle ne l'est pas. C'est l'instruction suivante, `invokespecial`, qui le sera ensuite. Elle est en particulier responsable de l'exécution des constructeurs classes.

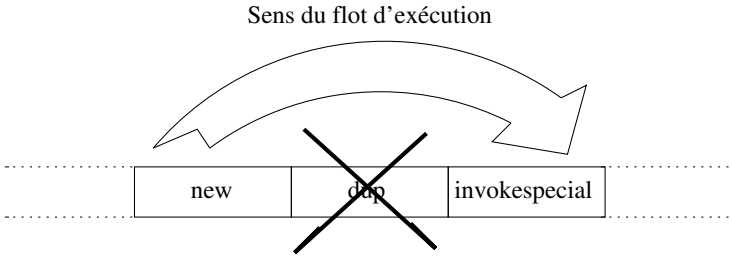

FIG. 7.9 – Flots d'exécution modifié de l'instruction `new`

Cependant, contrairement au schéma d'exécution standard (figure 7.10), ce n'est pas un composant *instruction* implantant `invokespecial` qui est sollicité.

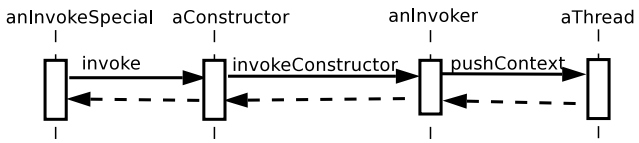

FIG. 7.10 – Appel standard d'un constructeur

L'exécution du constructeur est déléguée à la machine virtuelle *hôte*. Le mécanisme *Down* est ainsi utilisé pour le transfert des paramètres de la pile de Corosol vers sa pile. Le constructeur de la classe est appelé *via* réflexivité, de part l'utilisation d'un objet `java.lang.reflect.Constructor`. L'instance qui en résulte est ensuite convertie en un objet du niveau méta 0, *via* notre mécanisme *Up* (figure 7.11).

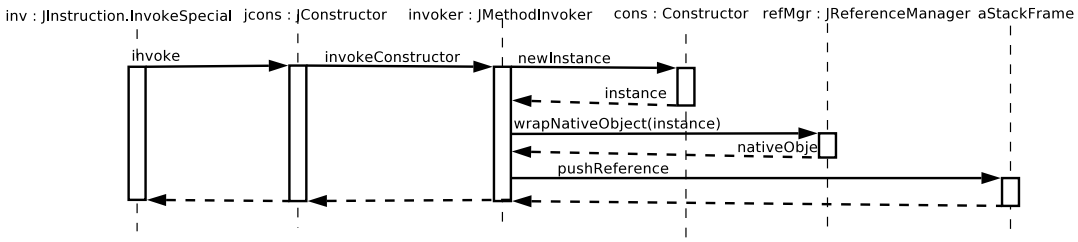

FIG. 7.11 – Appel d'un constructeur natif

7.4 Conclusion

Ce chapitre a détaillé les mécanismes réflexifs destinés à rendre possible la cohabitation des objets de l'application et ceux des méta-niveaux inférieurs, c'est-à-dire ceux de la machine virtuelle Corosol et de la machine virtuelle hôte. Nous avons montré comment cette application peut examiner et utiliser les composants de Corosol depuis le niveau de base. Nous avons abordé cela en expliquant le mécanisme de délégation aux méta-niveaux inférieurs lorsqu'un niveau ne sais pas comment interpréter un objet. Le chapitre suivant reviendra plus longuement sur la construction des *proxys*, car ils permettent d'implanter une partie majeure de ces mécanismes réflexifs dits *up/down* décrits dans [Ste94, WD01].

Chapitre 8

Construction des *proxys*

8.1 Introduction

Les *proxys* sont utilisés dans la gestion des composants de Corosol afin de faciliter leur remplacement. Ils sont aussi utilisés dans les mécanismes réflexifs de notre machine virtuelle, en particulier lorsqu'il s'agit de déléguer l'exécution à la machine hôte. Dans ce cas, ils encapsulent une instance de classe du niveau méta 0 au sein d'une instance native du niveau méta 1 et relaient les accès en lecture et en écriture entre ces deux instances. Ce sont ces notions que nous développerons dans ce chapitre. Nous montrerons aussi comment le *bytecode* Java de ces *proxys* est généré à la volée. Nous utiliserons pour cela les notions abordées au chapitre 3 concernant le *bytecode* Java.

8.2 Les *proxys* de composants

À l'ajout d'un nouveau type de composant dans l'architecture de notre machine virtuelle, un nouveau *proxy* est créé à la volée. Il est comparable à une «enveloppe» qui encapsule un composant concret. Il est d'ailleurs du même type que le composant qu'il encapsule : il possède en particulier les mêmes méthodes.

La construction des *proxys* de composants ne repose pas sur l'utilisation de la classe standard *java.lang.reflect.Proxy*, car celle-ci nécessite que les *proxys* ne soient générés qu'à partir d'une interface. Or, nous désirons aussi qu'ils le soient à partir d'une classe concrète. En effet, nous ne voulons pas imposer l'écriture d'une interface pour chaque nouveau composant : il peut ainsi avoir pour type abstrait son type concret. Examinons cette construction.
Soit `c` une instance d'un composant de type abstrait `C`.
Soit `CProxy`, la classe du *proxy* de ce composant, construite suit :

- `CProxy` hérite du type `C` ;
  Important : ce sera un héritage de classe **ou** d'interface, suivant que `C` soit elle-même une classe **ou** une interface) ;
- `CProxy` hérite également de l'interface `JProxy`, qui identifie toute les classes de `proxy` au sein de Corosol ;
- une variable d'instance nommée `component` lui est attribuée (ce nom est commun à toute les classes de *proxy* à leur création). Celle-ci désigne le composant encapsulé. Le type statique de cette variable d'instance est le type `C` et sa valeur est la référence désignée par `c`.

Ainsi, la classe de *proxy* `CProxy` possède les méthodes du type `C`. Le corps de chacune d'entre elles est un appel à la méthode du même nom sur la variable d'instance `component`. De plus, `CProxy` hérite aussi des méthodes de l'interface `JProxy`. L'implantation des méthodes de cette dernière est identique pour tous les *proxys* générés. La figure 6.7 du chapitre 6 les illustre :

- `getObject`, qui retourne l'objet encapsulé par le *proxy*,
- `setObject`, qui attribut l'objet spécifié comme nouvel objet encapsulé par le *proxy* ; cette méthode est utilisée lors du remplacement de composant.

Elle montre également comment les appels de méthodes de la classe `CProxy` sont délégués vers des appels aux méthodes sur la variable d'instance `component`, le composant de type `C` encapsulé par ce *proxy*. Dans l'exemple développé, le type de composant `C` possède au moins deux méthodes `method1` et `method2`. Lors de l'invocation de la méthode `method1`, un appel à la méthode `method1` de la classe `C` est effectuée sur l'instance `component` représentant le composant concret.

Les *proxys* de composants sont créés à la volée par génération de *bytecode* Java. Nous expliquons dans la suite la structure de ce code et détaillons les éléments de l'architecture de Corosol qui en sont responsables.

8.2.1 Création d'une classe de *proxy*

Comme dit précédement, chacune des classes de *proxy* est créée à la volée, lors de l'ajout d'un nouveau type de composant dans Corosol. Cela s'effectue en générant le *bytecode* Java associé *via* les éléments internes utilisés pour réifier, lors de l'exécution, chaque partie logique d'un fichier au format `class`. Le chapitre suivant illustrera leur utilisation lorsqu'il s'agira de représenter les entrées de la table `constant_pool`). Abordons maintenant la structure de ce *bytecode*.

8.2.1.1 Créer la table `constant_pool`

Comme expliqué au chapitre 3, le *bytecode* Java est sous la forme d'un fichier au format `class`. La partie la plus importante de ce dernier est la table `constant_pool`, plus simplement appelée *constant pool*. Le *constant pool* regroupe en particulier les différentes chaînes de caractères représentant les appels de méthodes et les accès aux champs.

La classe de *proxy* et le type du composant qu'elle encapsule possède les mêmes méthodes. L'implantation des méthodes de cette classe de *proxy* est simple. Le corps de chacune des méthodes est un appel vers la méthode du même nom mais située sur le type du composant encapsulé par ce *proxy*. Ainsi le *constant pool* sera composé des entrées correspondant à ces différents appels de méthode. Ces entrées seront de type `ConstantInterfaceMethodref_info` ou `ConstantMethodref_info`, suivant que le type abstrait du composant encapsulé par le *proxy* soit une interface ou une classe.

En plus des entrées représentant les appels de méthodes du composant encapsulé, le *constant pool* contient aussi les entrées correspondant à leur nom et à leur signature. Elles sont de type `ConstantString`. De la même manière, les classes de chacun des paramètres de ces méthodes sont aussi décrites en termes d'entrées du *constant pool* : elles sont de type `Constant_Class_info`.

Nous aurons compris qu'appeler une méthode sur une classe de *proxy* provoque l'appel à la méthode de même nom sur une instance de composant, en l'occurence celui encapsulé par ce *proxy*. Dans sa construction, une classe de *proxy* possède une variable d'instance désignant un tel composant. L'accès à un tel champ au cours de l'exécution est aussi représenté par une entrée du *constant pool* : elle est de type `Constant_Fieldref_info`.

Exemple Illustrons les différentes notions que nous venons d'aborder. Soit `A` un type de composant de Corosol, notre machine virtuelle. Soient deux méthodes `E m1(B b, C c)` et `F m2(D d)` définies par le type `A`. Lors de l'ajout d'un composant de type `A` dans Corosol, une classe de *proxy* est dynamiquement créée. Elle se nomme `AProxy` et est construite de la façon suivante :

- elle hérite de `A` et de `JProxy` ;
- elle possède une variable d'instance nommée `component` de type abstrait `A` ;
- elle se compose des méthodes héritées de `A`, savoir `m1` et `m2`, mais aussi,
- des méthodes de `JProxy`, à savoir, `getObject` et `setObject`.

Le code des méthodes de `AProxy` est généré automatiquement à la volée. Les séquences d'instructions de ce *bytecode* Java est simple. Nous examinons cela dans la suite.

8.2.1.2 Créer le code des méthodes

Reprenons l'exemple précédent. Un appel à une méthode de la classe de *proxy* `AProxy`, héritée du type `A`, provoque l'appel de la méthode du même nom sur l'objet contenu par sa variable d'instance `component`. Cet objet est de type `A`. Ainsi chacune des méthodes héritées posssède pour code Java, un code basé sur le modèle suivant (la méthode `m1` est prise comme exemple) :

```
E m1(B b, C c) {
  return component.m1(b, c);
}
```

La compilation en `bytecode` Java d'une telle méthode nécessite cinq étapes successives :

1. générer l'unique instruction qui empile la référence du *proxy*, c'est-à-dire la référence de l'objet qui appelle cette méthode (c'est-à-dire le `this` utilisé dans un programme source en Java) ;
2. générer l'unique instruction qui empile de la référence de l'objet contenu par la variable d'instance `component` ;
3. générer les instructions qui empilent la valeur des arguments de la méthode (dans leur ordre d'apparition dans la signature de celle-ci) sur la pile d'opérandes de la `frame` qui sera allouée pour l'exécution de la méthode (*Cf.* chapitre 4). Celles-ci serviront comme arguments d'appel de la méthode du composant encapsulé ;
4. générer l'instruction qui effectue un appel à la méthode du même nom sur le composant encapsulé et désigné par la variable d'instance `component` ;
5. générer l'instruction de gestion de la valeur de retour de cette méthode si celle-ci retourne quelque chose.

L'étape 1 est la plus facile à réaliser. L'objectif est d'empiler la référence d'un objet sur une *frame*, en l'occurence la référence de l'objet appelant, c'est-à-dire le *proxy* lui-même. Elle se traduit par l'unique instruction `aload_0`.

L'étape 2 est aussi simple. Il s'agit d'empiler la référence de l'objet contenu par la variable d'instance `component` du *proxy*. Une telle manipulation s'effectue *via* l'instruction `getfield` avec comme opérande l'indice de l'entrée du *constant pool* correspondant à `component`. Cet indice est identique pour toutes les méthodes de la classe de *proxy*. Celui-ci désigne donc une entrée de type `ConstantFieldref_info` qui décrit la variable d'instance `component` de ce *proxy*.

L'étape 4 est aussi immédiate, l'instruction `invokeinterface` exécutant les opérations désirées. Seules les étapes 3 et 5 nécessitent des manipulations supplémentaires.

L'étape 3, qui consiste à générer les instructions qui empilent chacune des valeurs des arguments de la méthode sur la *frame* courante, tient compte de ce que nous avons expliqué au chapitre 4 : les instructions de la machine virtuelle Java sont fortement typées, et en particulier celles manipulant les *frames*, de telle sorte qu'un type primitif Java est associé à une famille d'instructions. Ainsi les instructions générées seront toutes de la forme *T*`load`, *T* représentant le type de la valeur empilée. Chacune des instruction possède un operande calculé en fonction de la position de cette valeur dans le tableau des variables locales de la *frame* appelante (c'est-à-dire allouée pour la méthode qui appelle la méthode de *proxy*). Si la valeur est contenue dans la première case de ce tableau, l'indice sera 0. Si celle-ci est située dans la seconde case, l'indice sera 1, et ainsi de suite.

L'étape 5 suit la même démarche que l'étape 3. Il s'agit là encore de déterminer le bon type d'instruction en fonction d'un type primitif. Dans ce cas, il s'agit des instructions *T*return, *T* représentant le type de la valeur retournée par la méthode de *proxy*. D'ailleurs, si celle-ci ne retourne pas de valeur, ce type peut aussi être aussi `void` et dans ce cas l'instruction générée sera simplement `return`.

À titre d'exemple, considérons la méthode suivante. Elle appartient à la classe concrète d'un des composants de Corosol. Il s'agit de la méthode `allocateClassInstance` du type de composant `JHeap` implantée au sein de la classe `DefaultJHeap` :

```
public class DefaultJHeap implements JHeap {
...
  public JClassInstance allocateClassInstance(JClass c) {
    ...
  }
...
}
```

Nous désirons générer une classe de *proxy* possédant un objet `JHeap` comme une variable d'instance et dont chaque méthode délègue son exécution à la méthode du même nom sur cet objet. Ainsi pour la méthode `allocateClassInstance` de notre exemple, nous désirons générer le *bytecode* Java correspondant à ce qui suit :

```
public class JHeapProxy implements JProxy, JHeap {
```

```
  private JHeap component; //variable d'instance désignant le composant
...
  public JClassInstance allocateClassInstance(JClass c) {
    //délégation de l'exécution au composant encapuslé
    return component.allocateClassInstance(c);
  }
...
}
```

Dans les faits, nous obtenons ce qui suit (les opérandes de `getfield` et `invokeinterface` ne sont donnés qu'à titre d'exemple) :

```
aload_0        //on empile le this, i.e. la référence du proxy
getfield  #10  //on empile la valeur de la variable d'instance du proxy
aload     1    //la réference de c est empilé
invokeinterface #14,  2 //appel de component.allocateClassInstance(c)
areturn        //retour de la référence retournée par l'appel précédent
```

Nous venons d'aborder les *proxys* de composants et leur création au cours de l'exécution. Dans la section suivante, nous examinons les *proxys* d'instances de classes.

8.3 Les *proxys* d'instances

Lors de l'exécution, la machine virtuelle sous-jacente qui exécute Corosol peut être mise à contribution. Ce cas survient par exemple lors de l'exécution de méthodes natives (les méthodes de composants en sont un exemple). C'est dans ce contexte que nous servirons les *proxys* d'instances.

Au niveau méta 0, les instances de classes créées durant l'exécution de l'application sont représentées sous la forme d'objets de type `JClassInstance`. Les tableaux quant à eux sont représentés par des instances de la classe `JArray` (voir figure 8.1).

Cependant, une méthode native ne peut s'exécuter au niveau méta 0. Seul le niveau méta 1 en est capable. Les arguments de la méthode à exécuter doivent être alors des objets du niveau méta 1. Or ceux-ci proviennent du méta-niveau supérieur, le méta niveau 0 : ils ne sont pas du même type que ceux attendus par la méthode, car ils sont sous la forme d'instances des classes `JClassInstance` ou `JArray`. Il faut donc transformer chacun des objets de ces types en des objets du bon type, c'est-à-dire ceux correspondant à chacun des types décrits dans la signature de cette méthode.

Illustrons cela par un exemple. Soit la méthode native suivante :

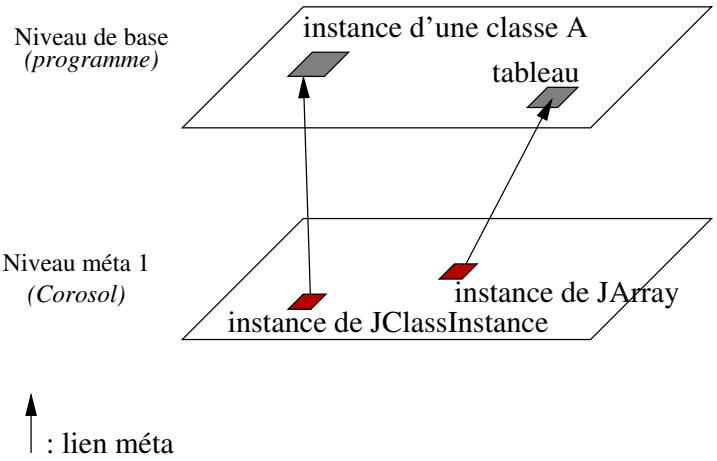

FIG. 8.1 – Représentation des objets du niveau de base au niveau méta 0.

```
native C m(A a, B b, int[] tab, float f).
```

Selon la signature de la méthode `m`, les arguments attendus sont des types `A`, `B`, `int[]` (tableau d'entiers) et `float`. Or, les objets de type `A` et `B` sont représentés au niveau méta 0 sous la forme d'objets `JClassInstance`. Le tableau est quant à lui représenté par une instance de la classe `JArray`. L'argument primitif de type `float` ne pose quant à lui aucun problème de représentation quel que soit le niveau. Ce sont des objets de ces types qui sont attribués en tant qu'arguments de la méthode `m`. Il existe donc un conflit entre ces types, à savoir `JClassInstance` et `JArray`, et les types attendus, savoir `A`, `B` et `int[]`.

Pour résoudre ce conflit de types, nous choisissons d'encapsuler chacun des objets du niveau méta 0 en un objet du niveau méta 1 qui possède le type attendu. Ainsi dans l'exemple précédent, l'instance de la classe `JClassInstance` qui représentait le type `A` est encapsulée dans un objet de type `A`, lors de son passage en tant qu'argument à la méthode native `m`. Il en est de même avec l'objet de type `B`. L'instance de la classe `JArray`, quant à elle, est transformée en un tableau natif (un tableau Java) du type attendu : ce sera un tableau de type `int[]`. C'est dynamiquement que tous ces objets seront créés. Nous les appelons *proxys d'instance*.

Le tableau 8.1 donne un récapitulatif quant à la transformation des objets entre les différents niveaux, ce qui se passe lorsque l'exécution est déléguée à un niveau inférieur (en l'occurence du méta niveau 0 vers le méta niveau 1).

Avant d'examiner le cas des tableaux, examinons tout d'abord comment

Type au niveau de base	Type au niveau méta 0	Type au niveau méta 1
`A`	`JClassInstance`	`AProxy (A)`
`B`	`JClassInstance`	`BProxy (B)`
`int[]`	`JArray`	`int[]`
`float`	`float`	`float`

TAB. 8.1 – Récapitulatif de la représentation des objets en cas de passage entre les différents niveaux.

une telle transformation s'effectue dans le cas des instances de classes.

8.3.1 Cas des instances de classes

Abordons le cas de la transformation d'un objet du niveau méta 0 en un objet du niveau méta 1. Les objets que l'on encapsule dans ses *proxys* ne sont tous définis par une interface : la construction des *proxys* ne repose donc pas sur l'utilisation de la classe standard *java.lang.reflect.Proxy*, tout comme pour la construction des *proxys* de composants.

Soit `c` un objet du niveau méta 0 qui représente une instance de classe d'un type `C` du niveau de base. `c` est donc de type `JClassInstance`.

La classe de *proxy* construit pour cette classe sera le type `CProxy`. Elle est construite comme décrit ci-après :

- `CProxy` hérite de la classe `C` ;
- `CProxy` implante également l'interface `JProxy`, qui identifie toute les classes de *proxy* au sein de Corosol ;
- une variable d'instance lui est attribuée : elle se nomme `instance` et désigne l'instance de classe `c` de type `JClassInstance`.

Un aperçu de cette construction est donné à la figure 8.2.

Ainsi, la classe de délégation `CProxy` possède par héritage les méthodes du type `C`, mais aussi les méthodes de l'interface `JProxy`. Les méthodes de `JProxy` sont implantées de manière identique aux *proxys* de composants. Le *bytecode* Java des méthodes héritées de la classe `C`, par contre, est créé à partir de l'original. Seule les instructions d'accès aux variables d'instances sont modifiées. Examinons comment.

Au sein du *bytecode* d'une méthode, les accès aux différentes variables d'instances d'une classe sont effectuées par les instructions suivantes :

- `getfield`, pour les accès en lecture,
- `putfield`, pour les accès en écriture.

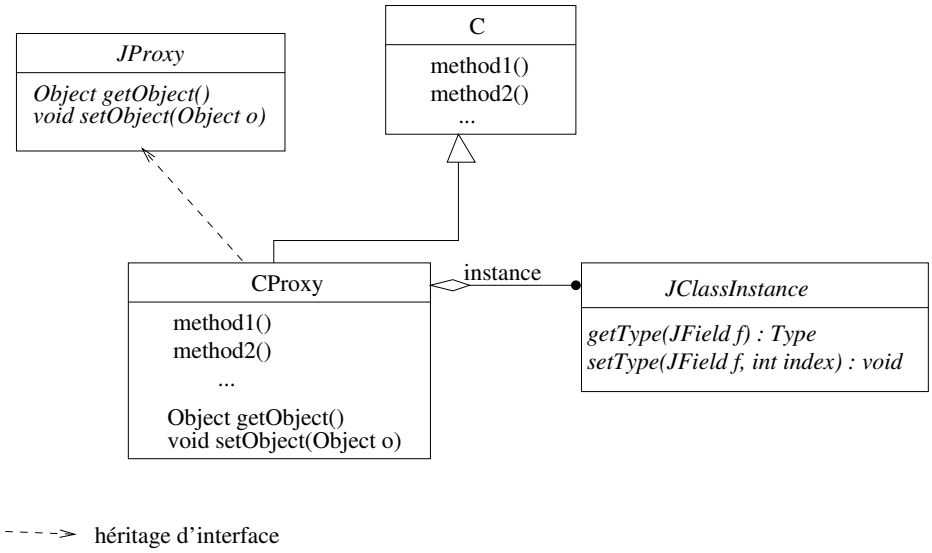

FIG. 8.2 – Proxys des instances de classes

L'interprète de Corosol exécute chacune de ces deux instructions pour accéder à la valeur d'une variable d'instance d'un objet du tas de celle-ci, ce dernier appartenant au niveau méta 0. Ces opérations sont effectuées à ce niveau par des objets qui appartiennent à celui-ci, c'est-à-dire des objets `JClassInstance`. De même, lors de l'exécution d'une méthode native au niveau méta 1, les variables d'instances modifiées ou accédées peuvent appartenir à des objets du tas de Corosol, du niveau méta supérieur, le niveau méta 0. Les mêmes objets `JClassInstance` doivent être aussi utilisés pour accéder ou modifier des valeurs du tas de celle-ci. C'est la raison pour laquelle ceux-ci sont encapsulés dans des *proxys* d'instances comme décrit à la figure 8.3.

La classe `CProxy` redéfinit donc chacune des méthodes de la classe `C`. Dans chacune d'entre elles, les instructions `getfield` et `putfield` sont remplacées par une séquence d'instructions permettant d'accéder en lecture et en écriture à une variable d'instance gérée *via* l'objet `JClassInstance` encapsulé par ce *proxy*.

Avant de décrire le contenu de cette séquence d'instructions, considérons la classe suivante :

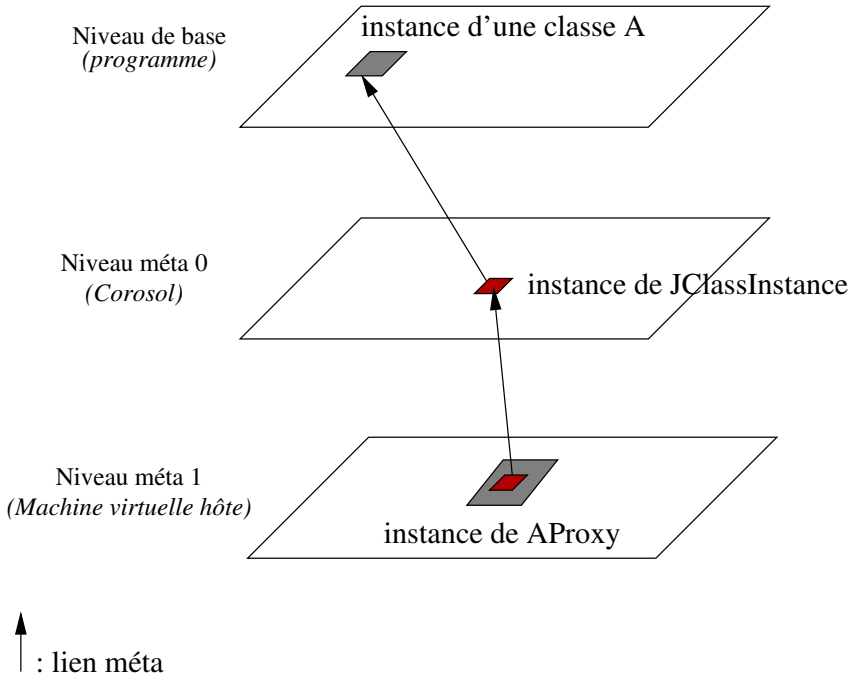

FIG. 8.3 – Proxys des instances de classes

```
public class JFieldHelper {
    static Type getType(JProxy proxy, int index);
    static void setType(JProxy proxy, Type value, int index);
}
```

Un objet de la classe `JProxy` est passé en tant qu'argument des méthodes de cette classe. Lors de l'exécution, ce sera un `proxy` d'instance qui encapsule un objet de type `JClassInstance`. L'état de ce dernier est respectivement consulté ou modifié *via* les méthodes `get`*Type* et `set`*Type*. Cela est effectué selon le type de la variable d'instance qui doit être mise à jour. Ce type est décrit par un index aussi passé en tant qu'argument. Il représente la valeur de l'opérande de l' instruction `getfield` ou `putfield` remplacée, qui est en fait l'indice d'une entrée dans le *constant pool* : celle-ci décrit le type de la variable d'instance à laquelle on accède. Ainsi, à partir de l'exécution d'une méthode d'un *proxy* d'instance au niveau meta 1, la valeur d'une variable d'instance peut être consultée ou modifiée dans le tas de Corosol (voir figure 8.4).

La séquence d'instructions des méthodes de la classe de *proxy* qui permet une telle chose est composée de ce qui suit, à savoir :

1. l'instruction qui réalise l'empilement d'un entier, celui-ci étant l'opérande du `getfield` ou du `putfield` initial : ce sera l'instruction `sipush` ;

2. l'instruction d'appel de méthode statique à `get`*Type* et `set`*Type* de la classe `JFieldHelper` : ce sera l'instruction `invokestatic`.

À titre d'exemple, considérons les méthodes `getX` et `setX` d'une classe `Point` :

```
public class Point {
  private int x;
  ...
  public int getX() {
    return this.x;
  }
  public void setX(int x) {
    this.x = x;
  }
...

}
```

La compilation *bytecode* Java de ces deux méthodes est décrit dans ce qui suit :

- pour `getX` :
  `aload_0`
  `getfield` *getfieldOp* //accès en lecture à la variable d'instance `x`
  `ireturn`
- pour `setX` :
  `aload_0`
  `iload_1`
  `putfield` *putfieldOp* //accès en écriture à la variable d'instance `x`
  `return`

Dans le cas de la classe de *proxy* générée pour une instance de la classe `Point`, le code surchargé des ces mêmes méthodes sera :

- pour `getX` :
  `aload_0`
  `sipush` *getfieldOp* //on empile l'ancien opérande de `getfield`
  `invokestatic` *index* //appel de la méthode `getInt` de `JFieldHelper`
  `ireturn`
- pour `setX` :
  `aload_0`
  `iload_1`
  `sipush` *putfieldOp* //on empile l'ancien opérande de `putfield`
  `invokestatic` *index* //appel de la méthode `setInt` de `JFieldHelper`

```
return
```

Remarque : Il est une idée à ne pas oublier. Lors de la construction du code du *proxy* d'une instance de classe ; chacun des opérandes des instructions de branchements doit être renuméroté. En effet, ajouter une nouvelle instruction au sein du *bytecode* d'une méthode entraîne un décalage dans celui-ci.

Nous venons d'aborder les proxys d'instance, leur structure et comment ils sont créés au cours de l'exécution. Dans la section suivante, nous examinerons comment un tableau du niveau méta 0 peut être transformé en un tableau natif du niveau méta 0.

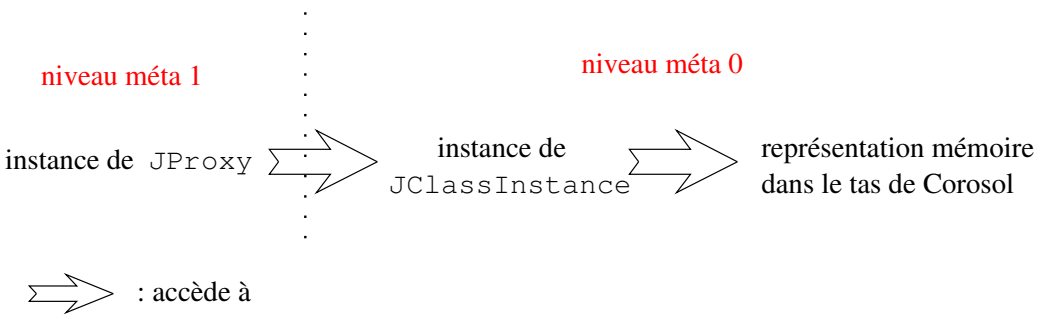

FIG. 8.4 – Utilisation des *proxys* d'instances

8.3.2 Cas des tableaux

Comme dit précédemment, les tableaux du niveau de base, celui de l'application, sont représentés au niveau méta 1 par des instances de la classe `JArray` (voir figure 8.5). Lorsque l'exécution d'une méthode est déléguée du niveau méta 0 au niveau méta 1 et que cette méthode possède un tableau comme argument, nous effectuons les opérations suivantes :

1. un tableau natif est créé à partir de la représentation de `JArray`, par recopie des valeurs contenues dans les différentes entrées ;
2. la méthode s'exécute au niveau méta 1 (par la machine virtuelle hôte) ;
3. de retour au niveau méta 0, l'exécution continue par recopie des valeurs de chaque entrée du tableau natif dans celles représentées par l'objet `JArray` précédemment utilisé.

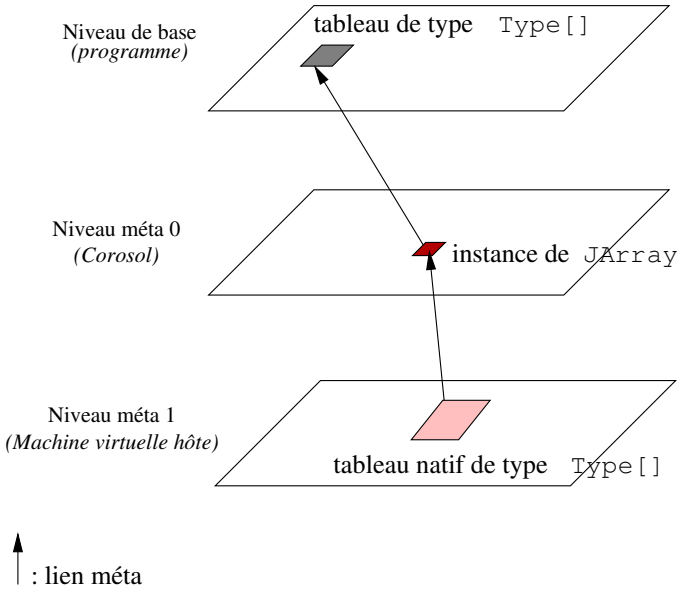

FIG. 8.5 – Proxys des tableaux

8.4 Conclusion

Ce chapitre nous a permis de décrire plus en détails le rôle et l'architecture des différentes classes de *proxys*. Deux catégories de ces classes sont à distinguer : les classes de *proxy* pour les composants et les classes de *proxy* pour les instances de classes. Nous avons souligné l'intérêt de telles structures lors du passage de l'exécution du niveau méta 1 (celui de Corosol) au niveau méta 0 (celui de la machine hôte qui exécute Corosol). En effet, un tel pont entre méta-niveaux est nécessaire lors de l'exécution de méthodes natives (et en particulier les méthodes des classes correspondant aux composants de Corosol). Chacune des classes de *proxy* est générée dynamiquement, à la volée. Ce chapitre a exposé la structure du *bytecode* des méthodes de ces classes. Les chapitres suivants donneront des exemples d'utilisation de ces mécanismes dans le cadre de l'adaptation de programme.

Quatrième partie

Adaptations de programmes avec Corosol

Chapitre 9

Adaptation avant l'exécution

9.1 Introduction

Ce chapitre montre un premier exemple de configuration de notre machine virtuelle. Nous décrivons comment l'ajout d'un nouveau type primitif au langage Java peut être supporté dans Corosol grâce à son architecture. Les opérations que nous effectuons pour cela sont réalisées avant le démarrage de notre machine virtuelle.

9.2 Ajout d'un type primitif : principe

Nous désirons exécuter du *bytecode* qui contient des informations relatives à un nouveau type primitif du langage de programmation Java. Plus précisément, nous définissons le nouveau type `complex` qui correspond à la représentation en Java d'un nombre complexe.

Pour que notre machine virtuelle puisse supporter ce nouveau type primitif, elle doit en premier lieu le reconnaître lors de son analyse du fichier au format `class` de l'application :

. au niveau du *constant pool* :

- ajouter un nouveau type primitif signifie ajouter une nouvelle entrée au *constant pool* (c'est en fait la table `constant_pool` que nous avons décrite au chapitre 3) ; dans un fichier au format `class`, la séquence d'octets correspondant aux données de cette nouvelle entrée est préfixée par une entête et celle-ci doit être reconnue durant l'analyse par notre machine virtuelle ;
- les entrées du *constant pool* correspondant aux appels de méthodes et à l'accès aux variables d'instance (ou de classe) contiennent le descripteur du nouveau type `complex` ; ce descripteur est une chaîne

de caractères qui identifie chaque type au sein de la machine virtuelle ; il doit pouvoir être reconnu par Corosol.

. au niveau des instructions : les instructions spécifiques au type `complex` doivent être reconnues ; pour ce qui est des autres instructions, rien ne change à l'exception de l'instruction `newarray` dont l'opérande, `atype`, est un nombre indiquant quel est le type primitif des éléments du tableau à créer au sein de la machine virtuelle ; dans le cas du type `complex`, la machine virtuelle doit pouvoir reconnaître correctement la valeur de cet opérande.

En second lieu, comme n'importe quelle autre machine virtuelle Java, Corosol doit attribuer une méta-classe au nouveau type `complex`. Pour le type `float`, cette méta-classe est `float.class`, qui est une instance de la classe `java.lang.Class`. Dans notre architecture nous représentons de telle classe par des instances dont les classes héritent de l'interface `JClass`. Afin d'ajouter le type `complex` pour qu'il puisse être supporté par Corosol, il nous faut aussi écrire une nouvelle classe héritant de cette interface. Cependant, comme nous le détaillerons dans la suite, le protocole à méta-objets de Corosol étant implicite, cette nouvelle classe (qui sera nommée `ComplexClass`) jouera un rôle primordial dans l'exécution liée à la manipulation des nombres complexes.

La dernière étape concerne la représentation en mémoire tant au niveau du tas de la machine virtuelle, qu'au niveau des `frames` au sein desquelles les méthodes s'exécutent. Il faut pouvoir paramétrer Corosol pour déterminer comment elle stockera les valeurs de type `complex` en mémoire.

9.2.1 Définir l'entrée du *constant pool*

Chaque type primitif possède un type d'entrée dans le *constant pool* d'un fichier au format `class`. Celle-ci contient la valeur d'une constante de ce type primitif. Ainsi, on définit l'entrée `CONSTANT_Complex_info` pour le nouveau type `complex`. La structure de cette nouvelle entrée est décrite par la figure 9.1.

`tag` est l'entête des entrées de type `CONSTANT_Complex_info` du *constant pool* d'un fichier au format `class`. Nous choisissons de lui attribuer la valeur `CONSTANT_Complex` que l'on décide de fixer à 13. Comme pour toutes les entrées du `constant pool`, l'entête `tag` est représentée sur un seul octet (`u1`).

`real` et `imaginary_unit` sont respectivement la valeur de la partie réelle et celle de la partie imaginaire d'un nombre complexe. Nous choisissons de représenter chacune de ces valeurs à l'aide de quatres octets (`u4`) stockés

```
CONSTANT_Complex_info {
  u1 tag;
  u4 real;
  u4 imaginary_unit;
}
```

FIG. 9.1 – L'entrée `CONSTANT_Complex_info`

selon l'ordre «gros boutiste»[1].

9.2.2 Définir un descripteur de type

Chaque type primitif de Java possède un descripteur qui est un caractère. Chaque descripteur représente un type au sein du fichier au format `class`. Par exemple, les caractères 'I' et 'J' sont respectivement les descripteurs des types primitifs «int» et «long». Ainsi, nous choisissons le caractère 'K' comme descripteur du nouveau type primitif `complex`.

9.2.3 Définir la méta-classe `complex.class`

La machine virtuelle Java associe une méta-classe (c'est-à-dire une classe qui représente une classe) à chacun des types primitifs du langage de programmation Java. Ainsi, `int.class` représente la classe associée au type primitif `int`. Elle est une instance de la classe `java.lang.Class`.

Pour le nouveau type primitif `complex`, nous devrons implanter une telle classe. Nous la désignerons par `complex.class`.

9.2.4 Définir la valeur `atype`

`newarray` est l'instruction de la machine virtuelle qui est chargée de la création d'un tableau de type primitif. Elle possède un unique opérande nommé `atype` dans la spécification. La valeur de cet opérande indique le type primitif des éléments du tableau à créer (*cf.* tableau 9.1).

newarray
atype

TAB. 9.1 – L'opérande `atype` de l'instruction `newarray`

[1] *big indian*

Par exemple, si `atype` possède la valeur 4, un tableau de `boolean` sera créé par la machine virtuelle. D'autres valeurs sont ainsi définies pour les autres types primitifs du langage Java. Nous décidons d'attribuer la valeur 12 à l'opérande `atype` pour le type primitif `complex`.

9.2.5 Définir les instructions

La plupart des instructions de la machine virtuelle Java ne manipulent qu'un seul type primitif. Par exemple, l'instruction `iadd` ne manipule que des valeur de type `int` : elle dépile deux valeurs de ce type depuis la *frame* courante puis y empile le résultat de leur l'addition, lui aussi du même type. L'instruction `dload` empile une valeur de type `double` sur la pile d'opérande d'une `frame`.

Pour ajouter le type primitif `complex`, il est nécessaire d'avoir les instructions suivantes qui existent aussi pour les autres types primitifs du langage Java :

- des instructions de manipulation des *frames*, comme l'instruction `dload` citée plus haut dans le cas du type `double` ; ces instructions permettent par exemple le transfert de valeur entre la pile des opérandes et le tableau des variables locales ;
- des instructions arithmétiques, à l'image de l'instruction `iadd` dans le cas du type `int` ; ces instructions effectueront exclusivement leurs opérations sur des valeurs de type `complex` ;
- des instructions de conversion du type `complex` vers un autre type primitif et inversement ;
- des instructions de comparaison de deux valeurs de type `complex`.

Puisque de nouvelles instructions sont nécessaires à la manipulation du nouveau type primitif `complex`, il faut aussi pouvoir les reconnaître au sein du *bytecode* d'une méthode. Dans un premier temps, il faut choisir un numéro unique pour chacune d'entre elles. Ce numéro est appelé *opcode*. L'analyseur de *bytecode* doit donc être aussi modifié pour pouvoir reconnaître ceux-ci ainsi que leurs opérandes.

9.2.6 Définir la représentation en mémoire

L'ajout d'un nouveau type primitif à Java suppose que les *frames* puissent stocker les valeurs de ce type. Chaque *frame* manipule des éléments de deux catégories :

- des éléments de catégorie 1, qui sont représentables sur une seule entrée de celle-ci ; ce sont par exemple les types primitifs `int` ou `float` (ou

des types représentant des valeurs plus petites comme `byte` ou `char`) ;
- des éléments de catégorie 2, représentables cette fois-ci sur deux entrées consécutives ; les types primitifs `long` et `double` sont de cette catégorie.

Une valeur de type `complex` contient la partie réelle et la partie imaginaire d'un nombre complexe. Nous choisissons donc de représenter une valeur de ce type par de deux entrées consécutives de catégorie 1 du méta-niveau 1. Une valeur de type `complex` sera donc un élément de catégorie 2.

Dans l'architecture de Corosol, la taille du type primitif `complex` est donc déterminée à partir de la taille des éléments de catégorie 1. Elle est égale au nombre d'octets des éléments de type `int` ou `float` au sein du composant *tas*. Pour pouvoir représenter des nombres complexes sur un plus grand nombre d'octets, la taille de ces éléments doit être augmentée *via* une modification de la représentation mémoire comme présentée au chapitre suivant. Cela vient du fait que toute la mémoire est centralisée par le *tas*.

9.3 Mise en œuvre avec Corosol

Maintenant, nous décrivons comment réaliser l'ajout du type primitif `complex` au sein de notre architecture de machine virtuelle. Il nous faut ainsi définir :

- une nouvelle entrée pour le *constant pool*,
- un nouveau descripteur de type,
- une nouvelle valeur de `atype`,
- de nouvelles instructions pour ce type,
- de nouvelles *frames*.

9.3.1 Définition de l'entrée `CONSTANT_Complex_info`

Notre architecture de machine virtuelle modélise chacune des entrées du `constant pool` d'un fichier `class` et en particulier celles représentant les types primitifs Java.

Il existe quatre types d'entrées correspondant aux types primitifs `int`, `float`, `long` et `double` (*cf.* tableau 9.2). Ces entrées représentent des constantes numériques utilisées dans le programme Java.

Dans notre architecture, chacune des entrées précitées est représentée par un élément interne, comme décrit à la figure 9.2. Ainsi, par exemple, le type `JConstantFloat` représente une entrée du *constant pool* de type `CONSTANT_Float_info`.

Pour ajouter une nouvelle entrée correspondant au type primitif `complex`, il faut créé un nouveau type Java. Ce sera un sous-type de `JLoadableConstant`.

Entrée	Type primitif
`CONSTANT_Integer_info`	`int`
`CONSTANT_Float_info`	`float`
`CONSTANT_Long_info`	`long`
`CONSTANT_Double_info`	`double`

TAB. 9.2 – Types primitifs et entrées du *constant pool*

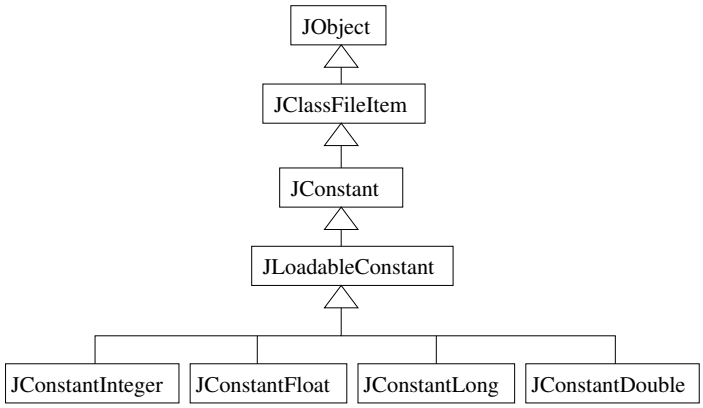

FIG. 9.2 – Modélisation des entrées de types primitifs

Cette interface Java décrit comment une entrée s'intègre au sein d'un *constant pool*. On distingue en particulier les méthodes suivantes :

- la méthode `getLength` retourne la taille de cette entrée dans le constant pool ;
- la méthode `getTag` retourne l'entier qui sert d'entête à cette entrée au sein du fichier au format `class` ;
- les méthodes `readItem` et `writeItem` qui réalisent respectivement la lecture et l'écriture des données de l'entrée depuis (respectivement vers) un fichier au format `class` ;
- la méthode `pushConstantValue` qui empile sur la pile des opérandes d'une *frame* la valeur de la constante numérique représentée par cette entrée ; le mécanisme de cette méthode sera en particulier utilisé par les instructions de la machine virtuelle `ldc`, `ldc_w` ou `ldc2_w` qui effectuent le transfert d'une valeur depuis le *constant pool* vers la pile des opérandes d'une *frame*.

Appliqué au type `complex`, nous implantons le nouveau type `JConstantComplex` que nous montrons à la figure 9.3.

```
public class JConstantComplex implements JLoadableConstant {
  public static int CONSTANT_COMPLEX = 13;

  private int re;
  private int im;

  public JConstantComplex() {
    this(0, 0);
  }

  public JConstantComplex(int re, int im) {
    this.re = re;
    this.im = im;
  }

  //retourne la taille de cette entrée au sein du
  //constant pool
  public int getLength() {
    return 2;
  }

  //retourne l'entête de cette entrée
  public int getTag() {
    return CONSTANT_COMPLEX;
  }

  //lecture depuis un fichier class
  public void readItem(JClassFileInput in)
    throws IOException
  {
    this.re = in.readInt();
    this.im = in.readInt();
  }

  //écriture dans un fichier class
  public void writeItem(JClassFileOutput out)
    throws IOException
  {
    out.write(CONSTANT_COMPLEX);
    out.writeInt(this.re);
    out.writeInt(this.im);
  }

  //chargement sur la pile des opérandes d'une frame
  public void pushConstantValue(JStackFrame frame) {
    frame.pushInt(this.re);
    frame.pushInt(this.im);
  }
}
```

FIG. 9.3 – La nouvelle entrée `JConstantComplex` en Java

9.3.2 Définition de `complex.class` et du descripteur

Dans notre architecture, le descripteur d'un type primitif est associé à l'implantation d'une méta-classe. Dans Corosol, chaque implantation de méta-classe est représentée par une instance d'une classe implantant l'interface `JClass` (*cf.* tableau 9.3). Cette interface possède en particulier des méthodes similaires à la classe `java.lang.Class`, comme celles permettant la recherche d'une méthode ou d'une variable d'instance ou de classe. Dans le cas des types primitifs, ces méthodes ont peu voire pas d'importance. En effet, la classe représentée par une méta-classe de type primitif ne possèdent ni champ ni méthode.

Considérons à titre d'exemple, le descripteur du type `long` qui est 'J'. La méta-classe associée à ce type au sein de la machine virtuelle Java est `long.class`. Dans Corosol, cette dernière implantée par une instance de la classe `LongClass`, qui est un sous-type de `JClass`. Le tableau 9.3 donne pour chacun des types primitifs de Java, le descripteur associé ainsi que l'implantation de sa méta-classe au sein de Corosol.

Puisque le protocole à méta-objets de Corosol est implicite (c'est-à-dire que Corosol l'utilise pour sa propre exécution), l'implantation de chacune de ces classes est primordiale.

Ainsi, un appel de méthode nécessite que la valeur de chacun des arguments soit empilée ou dépilée selon son type depuis la pile des opérandes d'une *frame*. Par exemple, si un argument est de type `int`, alors une valeur de ce type sera empilée ou dépilée. Ces mécanismes sont représentés par les méthodes `push` et `pop` de l'interface `JClass`.

Si la valeur des arguments d'une méthode est manipulée en fonction de

Descripteur	Méta-classe associée	Implantation Java dans Corosol
'B'	`byte.class`	`ByteClass`
'C'	`char.class`	`CharClass`
'D'	`double.class`	`DoubleClass`
'F'	`float.class`	`FloatClass`
'I'	`int.class`	`IntClass`
'J'	`long.class`	`LongClass`
'S'	`short.class`	`ShortClass`
'V'	`void.class`	`VoidClass`
'Z'	`boolean.class`	`BooleanClass`

TAB. 9.3 – Descripteurs et méta-classes

son type, il en est aussi de même pour la lecture et l'écriture de la valeur d'une variable d'instance. Chaque type primitif étant représenté par une instance d'une classe héritant de `JClass`, chacune de celle-ci fournie une implantation des méthodes `doGetfield` et `doPutfield` qui permettent de réaliser respectivement le mécanisme de lecture et d'écriture.

Appliquée au type `complex`, la méta-classe est une instance de la classe que nous avons appelée `ComplexClass`. Elle est présentée à la figure 9.4.

Les différentes associations entre un descripteur et l'implantation d'une méta-classe sont données au sein du fichier de configuration de Corosol. Pour le type primitif `complex`, nous avons choisi le caractère 'K' en tant que descripeur. L'association entre celui-ci et la classe `ComplexClass` doit aussi être ajoutée au sein de ce fichier de configuration. Il suffit d'ajouter la ligne suivante (`prim` est un préfixe facilitant l'analyse du fichier de configuration) :
`primK = ComplexClass`

9.3.3 Définition des instructions

Dans Corosol, chacune des nouvelles instructions est une instance d'une classe implantant l'interface `JInstruction`. La méthode `exec` de cette interface décrit comment une instruction s'exécute relativement à un processus léger. Celui-ci est représenté par une instance d'une classe implantant l'interface `JThread`. Elle possède en particulier les méthodes suivantes :

- `-getOperandInput,` qui permet d'obtenir un flot en lecture sur les opérandes de l'instruction ;
- `-getCurrentFrame,` qui retourne la `frame` du contexte d'exécution courant ; c'est sur au sein de celle-ci que seront empilés/dépilés les résultats de l'exécution de l'instruction, mais aussi la valeur des variables locales

```
public class ComplexClass extends JPrimitiveClass {
  public ComplexClass() {
    super("complex");
  }

  public void push(Object value, JStackFrame frame) {
    Complex val = (Complex)value;
    frame.pushInt(val.re);
    frame.pushInt(val.im);
  }

  public Object pop(JStackFrame frame) {
    Complex complex = new Complex();
    complex.im = frame.popInt();
    complex.re = frame.popInt();
    return complex;
  }

  public int getComputationalType() {
    return JClass.CATEGORY2;
  }

  public void doPutfield(JStackFrame frame, JField field) {
    Complex complex = new Complex();
    complex.im = frame.popInt();
    complex.re = frame.popInt();

    JClassInstance object = (JClassInstance)frame.popObject();
    object.setLong(field, (complex.im << 32 | complex.re));
  }

  public void doGetfield(JStackFrame frame, JField field) {
    JClassInstance object = (JClassInstance)frame.popObject();
    long value = object.getLong(field);

    frame.pushInt((int)(value >> 32));
    frame.pushInt((int)value);
  }

  public void doPutstatic(JStackFrame frame, JField field) {
     ...
  }

  public void doGetstatic(JStackFrame frame, JField field) {
     ...
  }

  public void arrayCopy(JArray src, int srcPos, JArray dest,
                        int destPos, int length)
  {
    for(int i = 0; i < length; i++) {
      long value = src.getLong(srcPos+i);
      dest.setLong(value, destPos+i);
    }
  }

  public void fieldCopy(JField field, JClassInstance in,
                        JClassInstance out)
  {
    long val = in.getLong(field);
    out.setLong(field, val);
  }
}
```

FIG. 9.4 – La méta-classe `complex.class`

utilisées, ce qui est réalisé avec les méthodes `load`*Type* et `store`*Type* de la classe `JStackFrame` ;

`-restoreContext,` qui dépile le contexte d'exécution courant afin de revenir au précédent ; cette méthode est utilisée par les instructions de type `return`, afin d'empiler un résultat sur la *frame* du contexte d'exécution précédent.

Quelques unes des instructions que nous avons implanté pour ajouter le type `complex` sont détaillées à la figure 9.5. Nous avons en particulier présenté :

- `complexadd` qui réalise l'addition de deux nombres complexes ; elle dépile deux valeurs de type `complex` et empile le résultat de leur addition ;
- `complexstore` qui, pour une *frame* donnée, effectue le transfert d'une valeur de type `complex` depuis la pile des opérandes vers une entrée du tableau des variables locales ; l'index de cette entrée est précisée en tant qu'unique opérande de cette instruction ;
- `complexload_1` qui effectue un transfert inverse, c'est-à-dire depuis une entrée du tableau des variables locales vers le sommet de la pile des opérandes ; l'index de l'entrée du tableau des variables locales est quant à lui implicite : il s'agit de l'entrée d'index 1 ;
- `complexreturn` qui empile sur la *frame* du contexte d'exécution précédent la valeur d'un élément de type `complex` qui est quant à lui situé

sur le *frame* du contexte d'exécution courant.

Dans chacune des implantations, chacune des *frames* manipule les valeurs de type `complex` en tant que deux valeurs de type `int`. Ainsi pour empiler une telle valeur, nous procédons à l'empilement d'un entier de type `int` correspondant à la partie réelle d'un nombre complexe, et d'un autre correspondant à sa partie imaginaire. Nous avons donc choisi de ne pas modifier la classe implantant l'interface `JStackFrame` qui représente la structure de la machine virtuelle du même nom.

```
//implantation de l'instruction complexadd
public class ComplexAdd implements JInstruction {
  public ComplexAdd() {
  }

  //exécute l'instruction au sein d'un processus léger
  public void exec(JThread thread) throws Throwable {
    JStackFrame frame = thread.getCurrentFrame();
    int im2 = frame.popInt();
    int re2 = frame.popInt();

    int im = frame.popInt();
    int re = frame.popInt();

    frame.pushInt(re + re2);
    frame.pushInt(im + im2);
  }

  //retourne le numéro de l'instruction
  public int getOpcode() {
    return Complex.COMPLEXADD;
  }
}
```

```
//implantation de l'instruction complexstore
public class ComplexStore implements JInstruction {
  public ComplexStore() {
  }

  //exécute l'instruction au sein d'un processus léger
  public void exec(JThread thread) throws Throwable {
    //lecture des opérandes
    JOperandInput in = thread.getOperandInput();
    int index = in.readUnsignedByte();

    JStackFrame frame = thread.getCurrentFrame();
    int im = frame.popInt();
    int re = frame.popInt();

    frame.storeInt(re, index);
    frame.storeInt(im, index+1);
  }

  //retourne le numéro de l'instruction
  public int getOpcode() {
    return Complex.COMPLEXSTORE;
  }
}
```

```
//implantation de l'instruction complexload_1
public class ComplexLoad_1 implements JInstruction {
  public ComplexLoad_1() {
  }

  //exécute l'instruction au sein d'un processus léger
  public void exec(JThread thread) throws Throwable {
    JStackFrame frame = thread.getCurrentFrame();
    int re = frame.loadInt(1);
    int im = frame.loadInt(2);
    frame.pushInt(re);
    frame.pushInt(im);
  }

  //retourne le numéro de l'instruction
  public int getOpcode() {
    return Complex.COMPLEXLOAD_1;
  }
}
```

```
//implantation de l'instruction complexload_1
public class ComplexReturn implements JInstruction {
  public ComplexReturn() {
  }

  //exécute l'instruction au sein d'un processus léger
  public void exec(JThread thread) throws Throwable {
    JStackFrame frame = thread.getCurrentFrame();
    int im = frame.popInt();
    int re = frame.popInt();

    //restauration du contexte précédent
    thread.restoreContext();

    frame = thread.getCurrentFrame();
    frame.pushInt(re);
    frame.pushInt(im);
  }

  //retourne le numéro de l'instruction
  public int getOpcode() {
    return Complex.COMPLEXRETURN;
  }
}
```

FIG. 9.5 – Quelques instructions manipulant le type complex

9.4 Mise à jour du fichier de configuration

Toutes les modifications que nous avons effectuées afin d'ajouter le type `complex` ne sont prisent en compte qu'avant le démarrage de Corosol. Notre architecture est informée des composants de base à instancier qu'après analyse d'un fichier de configuration. Celui-ci regroupe toutes les associations entre un type abstrait et un type concret. Ce dernier est chargé de réaliser l'implantation du type abstrait. Il existe des associations plus simple entre une chaîne de caractère (ou un nombre) et un type concret (c'était en particulier le cas lors de la prise en compte du nouveau descripteur du type `complex` - *cf.* section 9.3.2). Cela est particulièrement utile pour la description des différents types d'entrées du *constant pool* et pour la description des instructions de la machine virtuelle.

Nous donnons un extrait du fichier de configuration ci-après pour la prise en compte des nouvelles instructions manipulants les nombres complexex (le préfixe `opcode` n'est présent que pour faciliter l'analyse du fichier de configuration) :

```
opcode209 = ComplexStore
opcode210 = ComplexStore_0
opcode211 = ComplexStore_1
...
opcode222 = ComplexLoad_2
opcode223 = ComplexLoad_3
opcode230 = ComplexAdd
opcode240 = ComplexReturn
```

Pour chaque numéro d'instruction, ce fichier de configuration décrit l'implantation (c'est-à-dire le type concret associé).

Pour prendre en compte la nouvelle entrée du *constant pool*, à savoir `JConstantComplex`, on ajoute la ligne suivante au fichier :

```
13 = JConstantComplex
```

Comme dit précédemment, on ajoute le nouveau descripteur en décrivant l'association entre le caractère 'K' et la classe `ComplexClass`. Cela est rendu effectif par ajout de la ligne suivante au sein du fichier de configuration :

```
primK = ComplexClass
```

9.5 Conclusion

Dans cet exemple, nous avons montré comment configurer notre machine virtuelle Java, afin qu'elle puisse soutenir l'ajout d'un nouveau type primitif,

en l'occurence le type `complex`, correspondant à un nombre complexe. Nous avons pour cela tout d'abord identifié les composants à redéfinir et/ou à ajouter, puis nous en avons fourni une implantation simple pour chacun d'entre eux. Les modifications que nous avons apportées sont appliquées avant le démarrage de Corosol. D'autres modifications peuvent être apportées, comme nous l'avons expliqué dans les chapitres précédents, lors de l'exécution de celle-ci directement *via* l'application. C'est ce que nous verrons au chapitre suivant.

Chapitre 10

Adaptation durant l'exécution

10.1 Introduction

Le chapitre précédent a mis en évidence comment des changements peuvent être apportés avant l'exécution à l'architecture de Corosol pour la faire évoluer. Dans ce chapitre, nous traitons le cas des modifications apportées durant l'exécution, en l'illustrant par quelques exemples. Parmi eux, on expliquera comment changer des instructions, l'ordonnanceur ou encore le tas de notre machine virtuelle pendant son exécution. En dernier exemple, nous traitons le cas du changement dynamique du mécanisme d'appel de méthodes qui reposera sur un paquetage Java externe facilement intégré au sein de Corosol. Ces modifications sont ordonnées depuis l'application exécutée par notre machine virtuelle.

10.2 Accès aux composants de Corosol

Pour remplacer un ou plusieurs composants, une application a besoin d'accéder effectivement à ces derniers. Pour ce faire, elle utilise un point d'accès par l'intermédiaire duquel elle peut obtenir les différents composants de la machine virtuelle Java.

Dans notre architecture, `Corosol.getVirtualMachine()` est ce point d'accès. L'objet retourné est de type `JVirtualMachine` et représente effectivement la machine virtuelle qui exécute cette application. Considérons, par exemple, le code ci-dessous :

```
1   JVirtualMachine jvm = Corosol.getVirtualMachine();
2   JClassLoader loader =
3     (JClassLoader)jvm.getComponent(JClassLoader.class);
```

```
4   JClass c = loader.loadClass(args[0]);
5   System.out.println(c.getName());
```

Dans notre architecture, l'unique objet de type `JVirtualMachine` est un conteneur de composants. À partir de celui-ci, l'application peut obtenir l'ensemble des composants de la machine virtuelle Java qui l'exécute. Dans cet exemple, elle demande l'accès au chargeur de classes principal de Corosol (lignes 2-3). Après avoir obtenu une référence sur le composant qui le représente, c'est-à-dire un objet de type `JClassLoader`, l'application peut alors l'utiliser comme n'importe quel autre objet qu'elle pourrait créer. Cette portion de code effectue, par exemple, le chargement d'une classe (ligne 4) et en affiche le nom (ligne 5).

Un même procédé sera utilisé pour le remplacement des composants de Corosol. L'application demandera l'instance de la classe qui représente la machine virtuelle, et à partir de celle-ci elle pourra accéder et en modifier ses composants. L'accès aux composants sera effectué par la méthode `getComponent` du type `JVirtualMachine` (cela était illustré à la ligne 3 de la portion de code précédente). Leur ajout et leur remplacement seront réalisés respectivement par les méthodes `addComponent` et `replaceComponent`. Examinons maintenant comment réaliser ces opérations.

10.3 Remplacement dynamique d'un composant

La première étape du remplacement d'un composant lors de l'exécution d'une application consiste tout d'abord à écrire l'implantation concrète du nouveau composant. Son type abstrait est celui du composant à remplacer dans Corosol. Son type concret l'implantera au sens Java s'il est une interface ou en héritera s'il est une classe. Lors de cette étape, la méthode `replace` du nouveau composant doit être écrite si ce dernier doit être initialisé avec les valeurs de l'ancien composant. C'est important pour les composants comme le tas, qui sauvegardent des données de l'exécution.

La seconde étape consiste à récupérer le composant `JVirtualMachine` par la méthode statique `Corosol.getVirtualMachine`. Sa méthode `replaceComponent` réalise le remplacement d'un composant.

Ainsi, pour remplacer le composant de type abstrait `T` par sa nouvelle implantation `MyTImpl`, les opérations à réaliser par l'application sont les suivantes :

```
JVirtualMachine jvm = Corosol.getVirtualMachine();
jvm.replaceComponent(new MyTImpl());
```

La machine virtuelle recherchera le composant de type abstrait `T` et remplacera son implantation courante par la nouvelle instance `MyTImpl`. Pendant ces opérations, elle invoquera la méthode `replace` du nouveau composant.

Examinons maintenant quelques exemples de remplacements dynamiques de composants.

10.4 Remplacement d'une instruction

Nous commençons l'illustration du remplacement dynamique de composant par un exemple simple : le changement de la sémantique d'une instruction de *bytecode*.

10.4.1 Principe

Considérons l'instruction `iadd`. Celle-ci effectue l'addition de deux nombres entiers de type `int`, situés au sommet de la pile d'opérandes d'une `frame`. Pour ce faire, elle y dépile ces deux entiers, effectue leur addition, puis y empile ce résultat.

Remplacer une instruction de la machine virtuelle signifie modifier la sémantique de son exécution. Quelle que soit la modification apportée au comportement de cette instruction, il est donc imposé d'effectuer deux dépilements *élémentaires* et un empilement élémentaire. Par élémentaire, nous entendons des éléments de catégorie 1, dont les types possibles sont `int`, `float` ou des références (les types plus petits, `char`, `byte` et `boolean` sont transformés en `int` lors de leur passage au sein d'une *frame*). Les autres types d'éléments, à savoir `long` et `double`, sont de catégorie 2 et nécessitent, quant à eux, d'effectuer deux dépilements ou empilements élémentaires. En résumé, la nouvelle sémantique ne doit pas modifier la taille de la pile d'opérandes de la *frame* autrement que décrit dans la spécification de Sun Microsystems [LY99] sinon l'intégrité de l'exécution est remise en cause.

Modifions l'instruction `iadd` pour que celle-ci empile le produit des deux entiers dépilés de la pile d'opérandes. Elle devient alors comparable à l'instruction `imul` qui effectue les mêmes opérations de pile. Ainsi, les actions à réaliser sont les suivantes :

- dépiler le premier opérande,
- dépiler le second opérande,
- empiler le résultat de leur multiplication sur la pile.

La nouvelle instruction étant réalisée, il faut maintenant qu'elle soit prise en compte au sein de toute la machine virtuelle Corosol. Une instruction étant

elle-même un composant, la séquence d'opérations à effectuer est identique à celle de n'importe quel autre nouveau composant.

10.4.2 Mise en œuvre avec Corosol

Comme nous l'avons déjà mentionné, le type `JInstruction` modélise une instruction de la machine virtuelle Java. La méthode `void exec(JThread)` de ce type implante la sémantique de cette instruction dans Corosol. Ainsi, pour modifier le comportement de l'instruction `iadd`, il faut réécrire celle-ci. Commençons par examiner son implantation par défaut.

10.4.2.1 Implantation par défaut

Considérons la portion de code suivante :

```
1  public void exec(JThread thread) {
2    JStackFrame frame = thread.getCurrentFrame();
3    int value2 = frame.popInt();
4    int value1 = frame.popInt();
5    frame.pushInt(value1 + value2);
6  }
```

La ligne 1 met en évidence l'unique paramètre de la méthode `exec`, un objet de type `JThread`. Dans l'architecture de Corosol, il représente le fil d'exécution ou processus léger au sein duquel une instruction doit être exécutée. La ligne 2 montre que ce type possède en plus la méthode `getCurrentFrame()` qui permet d'obtenir la *frame* au sein de laquelle l'exécution se déroule. Chaque *frame* est modélisée par le type `JStackFrame`. Un objet de ce type est donc retourné par la méthode `getCurrentFrame`. Il possède les méthodes permettant de manipuler le tableau des variables locales et la pile des opérandes d'une *frame* et en particulier :

- `void pushInt(int value)` qui empile un entier de type `int`, au sens du niveau méta 1, sur la pile des opérandes et,
- `int popInt()` qui dépile un entier de type `int`, au sens du niveau méta 1, de la pile des opérandes et en retourne sa valeur.

Aux lignes 3 et 4, deux entiers sont successivement dépilés de la pile des opérandes de la *frame* obtenue précédemment (ligne 2). Finalement le résultat de leur addition est empilé sur celle-ci à la ligne 5.

Nous pouvons utiliser la structure de cette implantation par défaut comme modèle pour réécrire le comportement de l'instruction `iadd`.

10.4.2.2 Implantation de la nouvelle sémantique

Pour redéfinir la sémantique de l'instruction `iadd`, nous modifions la portion de code précédente. La ligne 5 est remplacée par `frame.pushInt(value1 * value2)`. La méthode `exec` s'écrit donc comme ci-après :

```
public void exec(JThread thread) {
  JStackFrame frame = thread.getCurrentFrame();
  int value2 = frame.popInt();
  int value1 = frame.popInt();
  frame.pushInt(value1 * value2);
}
```

L'intégralité du code de cette nouvelle instruction est donnée en exemple à la figure 10.1. Il est contenu au sein de la classe que nous avons nommé `MyIAdd` et qui implante l'interface `JInstruction.IAdd`, type abstrait du composant représentant l'instruction `iadd`. Celle-ci est une sous-interface de `JInstruction` et donc de `JVMComponent`. De part cette classe, nous montrons comment implanter la sémantique d'une instruction de *bytecode* du début jusqu'à la fin. Il est cependant possible de réaliser le nouveau comportement par dérivation de la classe d'instruction par défaut. Dans le cas de la nouvelle classe d'instruction `MyIAdd`, les méthodes de l'interface `JInstruction.IAdd` que nous devons implanter sont les suivantes :

- la méthode `configure`. Elle réalise l'intégration de l'instruction au sein de Corosol. Le corps de cette méthode est vide puisque le composant *instruction* `iadd` n'a pas de dépendance avec d'autre composant. Son intégration est directe.
- la méthode `getNativeObject`. Par son appel, on obtient l'objet du méta-niveau 1, c'est-à-dire celui de la machine virtuelle hôte, correspondant à ce composant. Cette méthode retourne l'objet instruction `MyIAdd` lui-même, car celui-ci est stocké à ce méta-niveau. Cela ce traduit par l'unique ligne suivante : `return this ;`
- la méthode `replace`. Cette méthode effectue le transfert de données d'un composant vers un autre du même type. Le composant *instruction* `iadd` ne sauvegarde pas d'état. Cette méthode possède donc un corps vide.
- les méthodes `getOpcode` et `toString`. Un appel de ces méthodes retourne respectivement un entier correspondant à la valeur du `bytecode` `iadd` et la chaîne de caractères représentant celui-ci.

Il reste maintenant à prendre en compte cette modification au sein de toute la machine virtuelle.

10.4.2.3 Intégration de la nouvelle instruction dans Corosol

Une instruction est un composant de Corosol et notre objectif est de remplacer le composant *instruction* `iadd` par sa nouvelle implantation. Ce remplacement de composant s'effectue donc *via* la méthode `replaceComponent` située sur le type `JVirtualMachine` et qui appelle la méthode `replace` sur tout nouveau composant. Dans le cas de la nouvelle implantation de l'instruction `iadd`, cette méthode ne réalise aucune opération car aucune information n'est sauvegardée par l'ancienne implantation. Considérons à nouveau la figure 10.1. Elle décrit l'application `Test3`.

Dans un premier temps, cette application effectue une addition de deux nombres entiers de type `int`. L'instruction de *bytecode* utilisée est l'instruction par défaut `iadd`. Le composant qui l'implante a pour type abstrait `JInstruction.IAdd`.

Dans un second temps, l'application demande la référence sur le dépôt des composants par `Corosol.getVirtualMachine()`. À partir de celui-ci, le remplacement de composant est réalisé : la méthode `replaceComponent` est utilisée, avec comme argument l'instance du composant `MyIAdd`, qui implante la nouvelle sémantique de l'instruction `iadd` :

```
JVirtualMachine jvm = Corosol.getVirtualMachine();
jvm.replaceComponent(new MyIAdd());
```

10.4.2.4 Exécution

Après avoir compilé le fichier `Test3.java` de la figure 10.1, celui-ci est exécuté par la commande suivante :

```
java fr.umlv.corosol.Corosol Test3 45 12
```

Dans celle-ci, on demande le calcul puis l'affichage de l'addition normale des nombres 45 et 12 puis de leur addition modifiée. On obtient en sortie ce qui suit :

```
Addition normale : 57
Addition modifiée en multiplication : 540
```

10.4.3 Atomicité et remplacement des instructions

Conformément au modèle d'exécution présenté au chapitre 6, l'exécution de chacune des instructions au sein d'un fil d'exécution est atomique. Ainsi, un tel processus léger ne peut être préempté que lorsque l'instruction qu'il exécute s'est totalement terminée.

```
import fr.umlv.corosol.component.*;
import fr.umlv.corosol.component.instruction.*;
import java.io.*;

public class MyIAdd implements JInstruction.IAdd {
  public MyIAdd() {}

  public Object getNativeObject() { return this; }

  public void configure(JVirtualMachine jvm) {}

  public void replace(JVMComponent component) {}

  public Class getComponentClass() {
    return JInstruction.IAdd.class;
  }

  public void exec(JThread thread) {
    JStackFrame frame = thread.getCurrentFrame();
    int value2 = frame.popInt();
    int value1 = frame.popInt();
    frame.pushInt(value1 * value2);
  }

  public int getOpcode() {
    return JInstruction.IADD;
  }

  public String toString() {
    return "iadd modifié";
  }
}
```

```
import fr.umlv.corosol.Corosol;
import fr.umlv.corosol.component.JVirtualMachine;

public class Test3 {
  public static void main(String[] args) {
    if(args.length != 2) {
      System.err.println("Usage : corosol Test3 <int value> <int value>");
      System.exit(1);
    }

    //lecture de deux nombres entiers puis addition normale
    int a = Integer.parseInt(args[0]);
    int b = Integer.parseInt(args[1]);
    System.out.println("Addition normale : " + (a+b));

    //On obtient une référence sur la JVM elle-même
    JVirtualMachine jvm = Corosol.getVirtualMachine();

    //remplacement de l'instruction d'addition par celle modifiée
    jvm.replaceComponent(new MyIAdd());

    System.out.println("Addition modifiée en multiplication :" + (a+b));
  }
}
```

FIG. 10.1 – MyIAdd.java et Test3.java

10.5 Remplacement de la représentation mémoire

Nous expliquons dans cette section comment la représentation de la mémoire de Corosol peut être remplacée à la volée depuis l'application exécutée.

10.5.1 Principe

Le composant *tas* est la mémoire de Corosol où sont contenus les objets de l'application, mais aussi les piles des fils d'exécution (voir chapitre 6). L'intérêt de remplacer le tas de la machine virtuelle est de pouvoir changer la représentation des objets de l'application et de lui donner par exemple la propriété de persistance. Il peut aussi permettre de modifier l'algorithme de ramasse-miettes de la machine virtuelle. Pour réaliser ce remplacement, il faut déplacer les objets *atteignables* de l'ancien tas vers le nouveau. Les objets atteignables sont ceux utilisables par l'application au moment du recyclage automatique de la mémoire par le ramasse-miettes, qui ne peut alors réclamer la mémoire qui leur a été attribuée. Pour effectuer le remplacement du composant *tas* de Corosol, il faut tout d'abord y reconnaître ces objets, mais aussi savoir traduire leur ancienne représentation dans celle du nouveau tas. La référence de chacun d'eux est contenue dans :

– le tableau des variables locales ou la pile des opérandes d'une *frame*

et/ou,
- la zone mémoire qui a été allouée pour un objet du tas, ou plusieurs ; dans ce cas elle la valeur de variables d'instances ou de classes.

10.5.1.1 Reconnaissance des objets atteignables

Il existe de nombreux algorithmes de ramasse-miettes permettant de déterminer l'ensemble des objets atteignables. La méthode par *comptage de références*, par exemple, associe un compteur à chaque objet. Il correspond au nombre de d'objets possédant une référence vers celui-ci. À la création d'un nouvel objet, son compteur est initialisé à zéro. Durant l'exécution, il est augmenté si la référence de l'objet devient la valeur d'une variable d'instance d'un autre objet et il est diminué si cette valeur est modifiée. Les objets atteignables sont ceux dont le compteur est supérieur à un.

L'algorithme précédent n'est pas efficace dans le cas des références circulaires, c'est-à-dire un ensemble d'objets qui se référencent mutuellement. Pour pallier à cela, il existe ainsi d'autres méthodes plus complexes, comme les algorithmes de traçage qui effectuent la recherche des objets atteignables par le parcours d'un graphe de références. Ce parcours commence par l'ensemble de références appelé *ensemble racine* (*root set*) constitué des références des objets contenues dans les piles des fils d'exécution et des références des variables de classes de l'application. Tous les objets rencontrés durant ce parcours sont atteignables.

10.5.1.2 Déplacement des objets atteignables

Les objets atteignables du composant *tas* à remplacer pouvant être reconnus, il faut maintenant pouvoir les déplacer vers le nouveau tas. Cependant, la copie octet par octet n'est pas satisfaisante, car la représentation des objets entre les deux tas peut être différente. Lors du déplacement, chaque objet voit sa position absolue modifiée. Cependant, ils conservent leur référence respective *via* l'utilisation du composant *gestionnaire des références* qui sépare la gestion des objets de la gestion de leur référence.

10.5.1.3 Déplacement des piles des fils d'exécution

Dans le modèle de mémoire de Corosol défini au chapitre 6, la mémoire attribuée aux piles des fils d'exécution est issue du composant *tas*. Ainsi, la représentation des valeurs qu'elles contiennent change et doit être traduite en celle du nouveau composant *tas*. La position absolue des piles peut éven-

tuellement être modifiée et dans ce cas, celle des *frames* qu'elles contiennent également.

10.5.2 Mise en œuvre dans Corosol

Nous expliquons maintenant notre mise en œuvre en Java du remplacement à la volée du composant *tas* de Corosol.

10.5.2.1 Implantation du nouveau tas

Le nouveau composant *tas* que nous désirons écrire réalise la persistance des données de l'exécution. Ses différents services sont implantés à l'aide de la classe `MappedByteBuffer` du paquetage `java.nio` qui représente un fichier entièrement transféré dans la mémoire de la machine virtuelle *hôte* (figure 10.2). Ce fichier est directement accessible sous la forme d'une séquence d'octets disponibles en lecture et en écriture qui représentera la mémoire de Corosol.

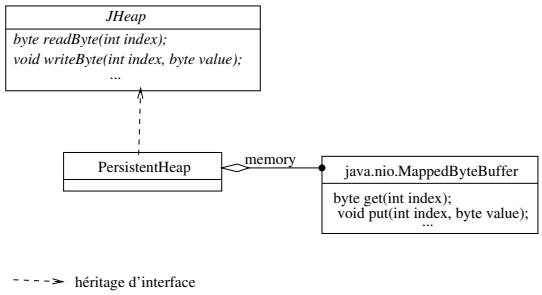

FIG. 10.2 – `PersistentHeap` : un composant *tas* persistent.

La figure 10.3 montre, par exemple, comment les services d'encodage et de décodage de ce nouveau *tas* sont réalisés dans le cas du type primitif `short`. `memory` y représente l'instance de la classe `MappedByteBuffer`, citée à la figure 10.2.

```
//décodage d'une valeur de type short
public short readShort(int index) {
  return (short)
    ((memory.get(index) << 8) |
     (memory.get(index + 1) & 0xff));
}
```

```
//encodage d'une valeur de type short
public void writeShort(int index, short value) {
  memory.put(index, (byte)((value >> 8) & 0xff));
  memory.put(index+1, (byte)((value) & 0xff));
}
```

FIG. 10.3 – Encodage et décodage dans le nouveau composant *tas*.

10.5.2.2 Déplacement des objets atteignables

Nous montrons maintenant comment mettre en œuvre le déplacement des objets atteignables. Pour ce faire, il faut commencer par mettre à jour les gestionnaires de dispositions de chaque classe.

Mise à jour des gestionnaires de dispositions : un *gestionnaire de disposition* est associé à chaque classe (voir chapitre 6). Il permet en particulier de déterminer les tailles des instances de classes et des tableaux au sein du tas de Corosol. La représentation de la mémoire étant modifiée, elles sont à recalculer.

Dans le cas d'une instance de classe, la taille de l'objet au sein de la mémoire correspond à la somme des octets nécessaires à la représentation de chaque champ et, dans le cas d'un tableau, à la somme des octets nécessaires à la représentation de chaque entrée. La taille de chaque type dépend de l'implantation concrète du composant *tas*. Elle est déterminée par son service `sizeof`.

La mise à jour d'un gestionnaire de disposition d'une classe d'objets s'effectue selon l'algorithme suivant, dans lequel sont calculés la taille de chaque instance de cette classe, mais également les positions relatives de ses variables d'instance au sein de leur zone mémoire respective :

Soit $newHeap$ le nouveau composant tas.
Soit $type$ le type de la classe associée au gestionnaire de disposition.
Soit $size$ la taille d'une instance de type $type$.

$c \leftarrow type$;
$size \leftarrow 0$;

Tant que $c \neq$ `null` **faire**
  Pour tout les variables d'instances $field$ de la classe c **faire**
    attribuer la position relative $size$ au champ $field$;
    $size \mathrel{+}= newHeap.sizeof(field.getType())$;
  FinPour
  $c = c.getSuperclass()$;
FinTantQue

Dans le cas des gestionnaire de disposition des classes de tableaux, la mise à jour s'effectue en redemandant la taille d'un élément au nouveau composant *tas*, par l'intermédiaire du service `sizeof`. La taille d'un tableau en octets

sera déterminée comme suit :

Soit *length* le nombre d'entrée du tableau.
Soit *type* le type de chacun de ses éléments.
Soit *newHeap* le nouveau composant tas.
Soit *size* la taille du tableau.

$size = newHeap.sizeof(type) \times length$

Recalcul des positions absolues des objets : maintenant que sont connues les tailles de chaque instance de classe et de chaque tableau, leur position absolue au sein du nouveau tas est recalculée par le composant `allocateur`.

Chaque objet Java de l'application est représenté par un objet de type `JClassInstance` ou `JArray`, respectivement pour les instances de classes et les tableaux. Ces deux types possèdent un super-type commun, `JAllocatable`, qui possède les trois opérations suivantes :

- `int getPosition()`, qui retourne la position absolue de l'objet dans le composant *tas*,
- `void setPosition(int position)`, qui attribue une nouvelle position absolue,
- `int getSize()`, qui retourne la taille de cet objet dans le composant *tas* où il est alloué.

Le composant *allocateur* manipule les objets alloués dans le tas par l'intermédiaire du type `JAllocatable`. Chacun des objets atteignables est de ce type. L'algorithme utilisé pour la recopie des objets dans le nouveau composant *tas* est le suivant :

Soit *freeOffset* l'indice du premier octet libre dans le composant *tas*.

$freeOffset \leftarrow 0$;
Pour tous les objets atteignables *object* **faire**
  $size \leftarrow object.getSize()$;
  $offset = allocate(size)$;
  $object.setPosition(offset)$;
FinPour

Recopies des valeurs des objets : il faut maintenant mettre à jour chacun des nouveaux objets du nouveau tas. Les valeurs des variables d'ins-

tances et les valeurs de chaque entrée de tableaux sont recopiées selon leur type. Par exemple, une valeur de type `int` sera lu en tant que `int` depuis l'ancien tas et recopié en tant que `int` dans le nouveau, la méthode de lecture et d'écriture dépendant de chaque tas. Le modèle de mémoire de Corosol sépare la représentation des références de leur gestion. Elles ne dépendent pas de la position d'un objet au sein de la mémoire. (voir chapitre 6). Les valeurs des références externes sont donc inchangées une fois transférées dans le nouveau composant *tas*.

La recopie des valeurs des différents objets se développe comme suit :

Pour chacun des objets atteignables *obj* de H_{old} :
- si *obj* est un tableau d'éléments de type T alors
  recopier selon le type T, les différentes valeurs de *obj* dans le nouveau tas H_{new} ;
- sinon *obj* est une instance de classe et
  recopier la valeur de chacune des variables d'instances selon son type dans le nouveau tas H_{new}.

Nous copions la valeur de chacune des variables d'instance selon son type déterminé par un objet `JClass`. Sa méthode `copyValue` permet de réaliser cette opération. Elle est définie pour chacun des types Java et permet de copier à une position absolue spécifiée la valeur d'un type primitif lue dans un tas donné dans un autre tas.

10.5.2.3 Déplacement des piles des fils d'exécution

Il faut désormais déplacer les piles des fils d'exécution car elles sont allouées au sein du composant *tas*, selon le modèle de mémoire de Corosol. Dans un premier temps, il faut recalculer la positions absolue de chaque *frames* contenues par ces piles. Dans un second temps, les valeurs des tableaux des variables locales et des piles des opérandes sont à recopier dans le nouveau composant *tas*.

Recalcul des positions absolues des *frames* : Chaque pile d'un fil d'exécution est de type `JavaStack`, dont `JAllocatable` est super-type. La taille d'une entrée des *frames*, c'est-à-dire la taille des éléments de catégorie 1 (voir chapitre 4), est égale à la taille d'une valeur de type `int` dans le nouveau composant *tas*.

Ainsi, la position absolue de chacune des piles de fil d'exécution, ainsi que celles des *frames* qu'elle contient sont recalculées selon l'algorithme suivant :

Soit $newHeap$ le nouveau composant *tas*.
Soit $stackPosition$ la position absolue de la pile dans $newHeap$.
Soit n le nombre de *frames* contenues dans la pile.
Soit $catSize$ la taille d'une entrée d'une *frame*.
Soit $getNbEntry$ le nombre d'entrées d'une *frame*.

$catSize \leftarrow newHeap.sizeof(JPrimitiveClass.INT)$
$JStackFramef \leftarrow stack.getStackFrame(0);$
$f.setCategory1Size(catSize)$
$f.setPosition(stackPosition)$

Pour i allant de 1 $n-1$ **faire**
$previous \leftarrow stack.getStackFrame(i-1);$
$current \leftarrow stack.getStackFrame(i);$
$position \leftarrow previous.getPosition() + catSize \times previous.getNbEntry();$
$current.setPosition(position);$
$current.setCategory1Size(catSize);$
FinPour

Recopies des valeurs des *frames* : Dans notre mise en œuvre du remplacement de la mémoire à la volée, nous avons choisi de typer la pile de chaque fil d'exécution de Corosol *avant l'exécution*, afin de permettre une recopie plus simple des données entre les deux piles. Le typage s'effectue par encapsulation de chaque élément interne *pile* dans une instance de type `TypedJavaStack`, correspondant à une pile dont la mémoire est typée (voir figure 10.4).

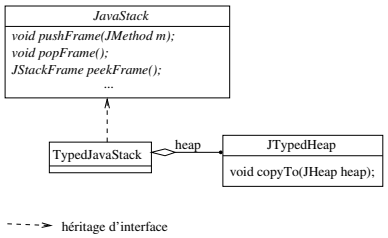

FIG. 10.4 – La pile typée d'un fil d'exécution.

L'élément interne *pile typée* est de type abstrait `TypedJavaStack` et possède le service particulier `copyValue`, permettant de recopier, *selon son type*, chaque valeur primitive qu'il contient au sein de la nouvelle pile passée en

argument. Par exemple, les valeurs de type `double` sont transférées comme suit :

Soit *tag* l'identifiant correspondant à un type primitif.

Soit *index* l'indice de lecture dans le tas initial typé.

Soit *destIndex* l'indice d'écriture dans le nouveau tas.

Soit *srcHeap* l'ancien composant *tas*.

Soit *destHeap* le nouveau composant *tas*.

Soit *sizeof* le service permettant d'obtenir la taille d'un type primitif décrit par sa méta-classe.

Soit *JPrimitiveClass.DOUBLE* la méta-classe du type `double`.

doubleValue ← srcHeap.readDouble(index) ;
destHeap.writeDouble(destIndex, doubleValue) ;
index += srcHeap.sizeof(JPrimitiveClass.DOUBLE) ;
destIndex += destHeap.sizeof(JPrimitiveClass.DOUBLE) ;

10.5.3 Récapitulatif

La figure 10.5 donne un extrait du code Java réalisant le remplacement du composant *tas* à la volée.

```
public void update(JHeap heap) {
  JVirtualMachine jvm = Corosol.getVirtualMachine();

  JLayoutFactory layoutFactory =
    (JLayoutFactory)jvm.getComponent(JLayoutFactory.class);
  layoutFactory.updateLayouts(heap);

  Iterator iterator = this.objects.iterator();
  this.freeOffset = 0;
  while(iterator.hasNext()) {
    JAllocatable next = (JAllocatable)iterator.next();
    int size = next.getSize();
    int offset = allocate(size);
    next.setPosition(offset);
  }
}
```

```
public void setPosition(int index) {
  this.position = index;

  if(isEmpty()) return;

  JVirtualMachine jvm = Corosol.getVirtualMachine();
  JHeap heap = (JHeap)jvm.getComponent(JHeap.class);
  this.category1Size = heap.sizeof(JPrimitiveClass.INT);
  JStackFrameImpl frame = (JStackFrameImpl)stack.get(0);
  frame.setCategory1Size(category1Size);
  frame.setPosition(this.position);

  for(int i = 1, size = stack.size(); i < size; i++) {
    JStackFrameImpl previous = (JStackFrameImpl)stack.get(i-1);
    JStackFrameImpl current = (JStackFrameImpl)stack.get(i);
    int pos = previous.getPosition();
    int size = previous.getNbEntry()*category1Size;
    current.setCategory1Size(category1Size);
    current.setPosition(pos + pize);
  }

  JStackFrame last = (JStackFrameImpl)stack.get(stack.size()-1);
  this.available = last.getPosition();
}
```

FIG. 10.5 – A gauche : implantation de la mise à jour des gestionnaires de dispositions et déplacement des objets. A droite : implantation de la mise à jour des positions absolues des *frames*.

10.6 Remplacement de l'ordonnanceur

Dans cette section, nous expliquons comment remplacer à la volée l'ordonnanceur de Corosol.

10.6.1 Principe

Dans Corosol, l'algorithme d'ordonnancement est indépendant de la gestion de la mémoire des fils d'exécution. Remplacer le composant *ordonnanceur* signifie alors modifier l'algorithme d'ordonnancement des fils d'exécution de Corosol.

La boucle d'exécution principale de Corosol prévoit l'arrêt de l'ordonnancement. Elle s'écrit selon l'algorithme suivant :

Tant que l'ordonnanceur n'est pas interrompu
  Ordonnancer les fils d'exécutions.
FinTantQue

Le modèle du composant *ordonnanceur* de Corosol n'assure pas encore la gestion des moniteurs et n'implante pas la politique de mémoire des fils d'exécution de la spécification de la machine virtuelle Java [LY99]. En plus de celles que nous décrivons, de nombreuses opérations supplémentaires doivent donc être initiées pour le changement à la volée de l'ordonnanceur de Corosol.

Dans notre cas, pour procéder au changement du composant *ordonnanceur* courant, il suffit tout d'abord de procéder à son arrêt. Les fils d'exécution qui ne sont pas terminés doivent être ensuite insérés dans la file du nouveau composant *ordonnanceur*. C'est ce nouveau composant qui initie l'arrêt du précédent.

10.6.2 Mise en œuvre dans Corosol

Dans l'architecture de Corosol, le type abstrait `JScheduler` représente le composant *ordonnanceur*, qui peut être arrêté *via* le service `breakScheduling`. `JThread` est le type des fils d'exécution.

Le remplacement du composant *ordonnanceur* s'effectue selon l'algorithme suivant :

Soit *oldScheduler* l'ancien composant *ordonnanceur*.

oldScheduler.breakScheduling();
Pour tous les fils d'exécution *thread* non terminés **faire**

ajouter *thread* dans la file du nouveau composant *ordonnanceur*;
FinPour

Le diagramme de séquences présenté à la figure 10.6 résume l'algorithme utilisé. La figure 10.7 détaille de code complet du nouvel ordonnanceur.

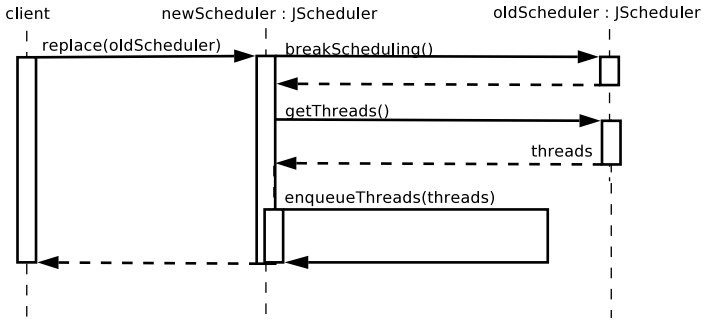

FIG. 10.6 – Remplacement du composant *ordonnanceur*

10.7 Ajout dynamique du *multi-dispatch*

Dans cette section, nous expliquons comment ajouter dynamiquement le mécanisme du multi-polymorphisme à notre machine virtuelle Corosol *via* l'usage du paquetage Java Multi-Method Framework (JMMF) [FDR00, For01] entièrement réalisé en Java. Cet exemple illustre la simplicité de l'intégration dans Corosol d'une bibliothèque *externe*.

10.7.1 Le *multi-dispatch*

Les langages à objets effectuent la sélection de la méthode à exécuter en fonction du type de chacun des arguments de celle-ci. Cette sélection, appelée *dispatch*, peut intervenir à deux moments particuliers :

- **au moment de la compilation** : c'est le *dispatch statique*;
- **au moment de l'exécution** : c'est le *dispatch dynamique*. Il permet aux langages objets de réaliser l'appel de méthode en fonction des types précis des objets connus au moment de l'exécution de l'appel.

Les langages à objets avec *dispatch* dynamique peuvent être ainsi divisé en deux catégories, suivant le nombre d'arguments considérés pour la sélection de la méthode à exécuter :

```
import fr.umlv.corosol.component.JClassMethod;
import fr.umlv.corosol.component.JScheduler;
import fr.umlv.corosol.component.JThread;
import fr.umlv.corosol.component.JVMComponent;
import fr.umlv.corosol.component.impl.AbstractJVMComponent;
import fr.umlv.corosol.repository.JImplementationRepository;

import java.util.LinkedList;

public class MyScheduler
  extends AbstractJVMComponent implements JScheduler
{
  private LinkedList threadQueue;

  public MyScheduler() {
    this.threadQueue = new LinkedList();
  }

  public JThread getCurrentThread() {
    return (JThread)threadQueue.getFirst();
  }

  public void enqueueThread(JThread thread) {
    threadQueue.addLast(thread);
  }

  public void schedule() throws Throwable {
    while(!threadQueue.isEmpty()) {
      JThread current = (JThread)threadQueue.getFirst();

      while(current.hasMoreInstructions()) {
        current.execNextInstruction();
      }

      threadQueue.removeFirst();
    }
  }

  public String toString() {
    return "MyScheduler : " + threadQueue.toString();
  }

public Class getComponentClass() {
    return JScheduler.class;
  }

  public void replace(JVMComponent component) {
    if(!(component instanceof JScheduler)) {
      throw new IllegalArgumentException();
    }

    if(component == this) {
      return;
    }

    JScheduler scheduler = (JScheduler)component;
    scheduler.breakScheduling();
    JThread[] threads = scheduler.getThreads();
    for(int i = 0; i < threads.length; i++) {
      enqueueThread(threads[i]);
    }
  }

  public void breakScheduling() {
  }

  public JThread[] getThreads() {
    return (JThread[])threadQueue.toArray(new JThread[0]);
  }

  public boolean hasNext() {
    return (!threadQueue.isEmpty());
  }
}
```

FIG. 10.7 – Le nouvel ordonnanceur `MyJScheduler`

- les langages à *uni-dispatch*, qui ne considèrent qu'un seul argument pour la sélection de la méthode. Celui-ci correspond au type concret de l'objet sur lequel cette méthode est appelée.
- les langages à *multi-dispatch*, qui considèrent plusieurs arguments, voire tous durant cette sélection.

Par exemple, Smalltalk est un langage à *uni-dispatch*, et CLOS et Cecil sont deux langages à *multi-dispatch*. Dans le langage Java (mais aussi C++), la sélection de la méthode à exécuter s'effectue lors de l'exécution et elle est déterminée en fonction du type d'un seul argument, ce qui correspond à une technique de *uni-dispatch dynamique*. La méthode choisie n'est déterminée qu'à partir du type de l'objet sur lequel cette méthode est appelée.

10.7.2 Mise en œuvre dans Corosol

Avant d'expliquer la réalisation du multi-polymorphime au sein de notre machine virtuelle, rappelons comment la sélection d'une méthode est effec-

tuée. Pour ce faire, examinons l'instruction `invokevirtual` et son implantation, le composant `Jinstruction.InvokeVirtual`.

10.7.2.1 Sémantique de l'exécution de `invokevirtual`

L'instruction `invokevirtual` est responsable de l'exécution d'une méthode. Selon la spécification de la machine virtuelle Java, elle effectue successivement les opérations suivantes :

- elle dépile les arguments de la méthode à exécuter, tant les paramètres que la référence correspondant à l'objet appelant (c'est-à-dire celui sur lequel la méthode est appelé) ;
- elle obtient la référence symbolique de la méthode à exécuter à partir du *constant pool* de la classe courante, c'est-à-dire la classe à partir de laquelle cette méthode est appelée ; cette référence symbolique est constituée des chaînes de caractères correspondant au nom de la méthode, sa signature et au nom de la classe de celle-ci ;
- elle effectue la résolution dynamique de la méthode à partir de la référence symbolique de celle-ci ; cette résolution a pour objectif de transformer cette référence symbolique en référence concrète, celle-ci correspondant à la méthode *la plus appropriée* sélectionnée à partir du type statique de l'objet appelant ;
- elle effectue une nouvelle recherche de méthode à partir des superclasses du type de l'objet appelant pour vérifier si elle n'a pas été redéfinie ; auquel cas, la méthode nouvellement trouvée est celle qui sera exécutée.

10.7.2.2 Le composant `JInstruction.InvokeVirtual`

Le composant `JInstruction.InvokeVirtual` réalise les différentes opérations définies par l'instruction `invokevirtual`. Il dépend dynamiquement des composants suivants :

- le composant `JClassLoader`, qui possède les services liés à la résolution des méthodes ;
- le composant `JMethodLookup` qui possède les services liés à la recherche d'une méthode dans une hiérarchie de classes.

La figure 10.8 montre le diagramme de séquences de l'implantation du composant `JInstruction.InvokeVirtual`. Le type concret de celui-ci est `InvokeVirtual`.

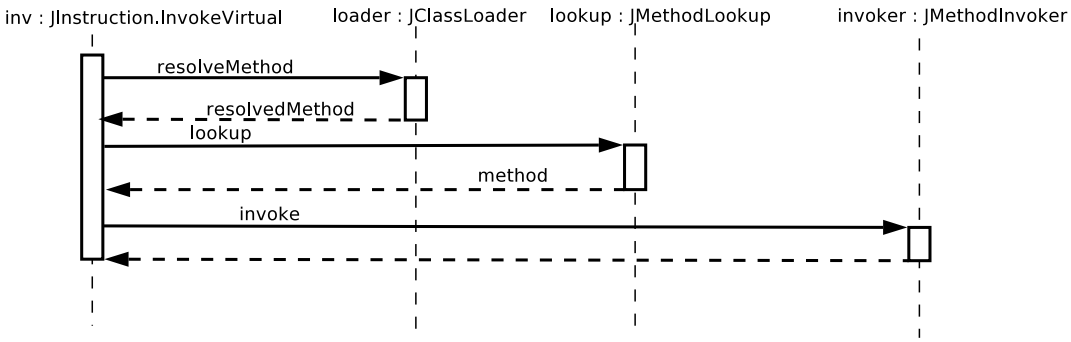

FIG. 10.8 – Diagramme de séquences `InvokeVirtual`

10.7.2.3 Le composant `JMMFInvokeVirtual`

Afin d'inclure la technique du *multi-dispatch* au sein de Corosol, nous avons réalisé le composant *instruction* de type `JMMFInvokeVirtual`, dont le type abstrait est `JInstruction.InvokeVirtual`, qui est le composant *instruction* réalisant les opération de l'instruction de *bytecode* `invokevirtual`.

Notre nouveau composant est construit grâce à la bibliothèque des multi-méthodes Java Multi-Method Framework (JMMF) [FDR00, For01] entièrement réalisée en Java. Dans celle-ci, les méthodes de la classe `MultiMethod` permettent d'effectuer le *multi-dispatch*. Dès lors les opérations à effectuer sont simples :

1. dans un premier temps, nous dépilons de la pile des opérandes de la *frame* courante tous les arguments d'appels de la méthode à exécuter, la référence de l'objet appelant comprise ;

2. dans un second temps, nous recherchons la méthode la plus précise *via* la méthode `getMethod` de la classe `MultiMethod`, à partir de son nom et des types dynamiques des arguments précédemment dépilés ; cette méthode est donnée sous la forme d'une instance de la classe `Method` du paquetage `java.lang.reflect` ;

3. dans un troisième temps, il ne reste plus qu'à exécuter *via* un appel réflexif la méthode précédemment sélectionnée par la technique de *multi-dispatch* ;

4. dans un quatrième temps, le résultat de l'exécution de la méthode est empilé sur la *frame* courante (c'est-à-dire celle sur laquelle ont été dépilés les arguments de cette méthode).

10.7.2.4 Exemple d'utilisation

Considérons la classe `JMMTest` présentée à la figure 10.9. Elle possède une méthode `m`, surchargée quatre fois et qui retourne une chaîne de caractères correspondant au type dynamique de ses arguments, qui peuvent être de type `A` ou de type `B`, sous-type de `A` :

- `m(A a1, A a2)` : retourne la chaîne `"AA"` ; les deux arguments sont des instances de la classe `A` ;
- `m(A a, B b)` : retourne la chaîne `"AB"` ; les deux arguments sont respectivement des instances de la classe `A` et de la classe `B` ;
- `m(B b, A a)` : retourne la chaîne `"BA"` ; les deux arguments sont respectivement des instances de la classe `B` et de la classe `A` ;
- `m(B b1, B b2)` : retourne la chaîne `"BB"` ; les deux arguments sont des instances de la classe `B`.

```
public class A {}

public class B extends A {}

public class JMMFTest {
  public String m(A a1, A a2) {
    return "AA";
  }

  public String m(A a, B b) {
    return "AB";
  }

  public String m(B b, A a) {
    return "BA";
  }

  public String m(B b1, B b2) {
    return "BB";
  }

  public static void main(String[] args) {
    A a = new A();
    A b = new B();
    JMMFTest test = new JMMFTest();

    System.out.println("Standard execution :");
    System.out.println(test.m(a, a));
    System.out.println(test.m(a, b));
    System.out.println(test.m(b, a));
    System.out.println(test.m(b, b));

    JVirtualMachine jvm = Corosol.getVirtualMachine();
    jvm.replaceComponent(new JMMFInvokeVirtual());

    System.out.println("JMMF execution :");
    System.out.println(test.m(a, a));
    System.out.println(test.m(a, b));
    System.out.println(test.m(b, a));
    System.out.println(test.m(b, b));
  }
}
```

FIG. 10.9 – La class `JMMFTest`.

Deux séries de tests sont effectués dans lesquels les différentes méthodes `m` sont appelées sur une instance de la classe `JMMFTest`. À chaque fois, la chaîne de caractères retournée par chaque méthode `m` est affichée. Dans chacun des appels, les deux arguments sont de type statique `A`. Le type dynamique du premier est `A` et celui du second est `B`.

La première série de tests est exécutée avec l'algorithme classique de résolution de méthodes, pour finalement aboutir à l'affichage suivant :

```
Standard execution :
AA
AA
AA
AA
```

Les quatre fois, la méthode `m(A a1, A a2)` a été exécutée, ce qui est le schéma d'exécution normal, le polymorphisme s'effectuant en fonction du type dynamique de l'objet appelant (`JMMFTest` dans notre cas) et non de ceux des arguments de la méthode appélée (dans notre cas, seul le type statique des objets, c'est-à-dire `A`, a été pris en compte).

Dans un second temps, on modifie l'instruction de *bytecode* `invokevirtual`, en remplacant le composant de type abstrait `JInstruction.JInvokeVirtual` par notre implantation concrète `JMMFInvokeVirtual`, afin d'intégrer le *multi-dispatch* dans Corosol. On exécute ensuite les mêmes méthodes `m` et on obtient :

```
JMMF execution :
AA
AB
BA
BB
```

Le *multi-dispatch* permettant de choisir la méthode à exécuter en fonction du type dynamique de tous les arguments, les différentes méthodes `m` ont été exécutées.

10.8 Conclusion

Ce chapitre a illustré par quelques exemples la mise en œuvre du remplacement dynamique de composant au sein de l'architecture de Corosol. Nous avons aussi discuté, dans chaque cas, du moment où est pris en compte telle ou telle modification dans l'exécution. L'exemple du changement d'une instruction, puis celui de l'ordonnanceur ont rappelé que, dans Corosol, que l'exécution d'une instruction est atomique, contrairement à la spécification de la machine virtuelle [LY99].

```
/**
 * Executes this instruction in the specified thread.
 *
 * @param thread a thread
 */
public void exec(JThread thread) {
  int index = thread.getOperandInput().readShort();
  JClass currentClass = thread.getCurrentClass();

  JConstantPool constantPool =
    currentClass.getConstantPool();

  JConstantMethodref constant =
    (JConstantMethodref)constantPool.getConstant(index)

  JClassMethod resolvedMethod = null;
  if(constant.isResolved()) {
    resolvedMethod = constant.getResolvedMethod();
  }
  else {
    resolvedMethod =
      loader.resolveMethod(currentClass,
                           constant.getName(),
                           constant.getClassName(),
                           constant.getDescriptor());
    constant.setResolvedMethod(resolvedMethod);
  }

  JStackFrame currentFrame = thread.getCurrentFrame();
  JHeapObject object =
    currentFrame.peekCallerObject(resolvedMethod);

  JClass declaringClass = object.getType();
  String name = resolvedMethod.getName();
  String desc = resolvedMethod.getDescriptor();
  JClassMethod method =
    methodLookup.lookup(declaringClass, name, desc);

  if(method == null) {
    method = methodLookup.superclassLookup(
               declaringClass, name,desc);
  }

  if(method == null) {
    throw new AbstractMethodError(
                resolvedMethod.toString());
  }

  method.invoke(invoker, thread);
}
```

```
/**
 * Executes this instruction in the specified thread by
 * using the multi-dispatch mecanism.
 *
 * @param thread a thread
 */
public void exec(JThread thread) {
  int index = thread.getOperandInput().readShort();
  JClass currentClass = thread.getCurrentClass();
  JConstantPool constantPool = currentClass.getConstantPool();
  JConstantMethodref constant =
    (JConstantMethodref)constantPool.getConstant(index);

  JClass[] paramTypes =
    getParameterTypes(constant.getDescriptor());

  JHeapObject[] popargs = new JHeapObject[paramTypes.length];
  Object[] args = new Object[paramTypes.length];
  Class[] nativeTypes = new Class[paramTypes.length];

  JStackFrame frame = thread.getCurrentFrame();

  //pop of the frame arguments
  for(int i = popargs.length-1; i >= 0; i--) {
    popargs[i] = paramTypes[i].pop(frame);
    args[i] = popargs[i].getNativeObject();
    nativeTypes[i] = args[i].getClass();
  }

  //pop of the this
  JHeapObject object = frame.popReference();
  Object target = object.getNativeObject();
  Class targetClass = target.getClass();

  //MultiMethod creation
  MultiMethod multiMethod =
    MultiMethod.create(targetClass, constant.getName(), args.length);

  //method call
  Method m = multiMethod.getMethod(targetClass, nativeTypes);
  Object returnValue = m.invoke(target, args);

  //push the result
  JClass returnType = getReturnType(constant.getDescriptor());
  returnType.push(returnValue, frame);

  //array copy...
  for(int i = 0; i < args.length; i++) {
    if(args[i] != null) {
      Class c = args[i].getClass();

      if(c.isArray()) {
        JArray src =
          (JArray)referenceManager.wrapNativeObject(args[i]);
        JArray dest = (JArray)popargs[i];
        JClass componentType = dest.getComponentType();
        componentType.arrayCopy(src, 0, dest, 0, src.length());
      }
    }
  }
}
```

FIG. 10.10 – Implantation de `invokevirtual` en *uni* et en *multi dispactch*

Cinquième partie

Conclusion

Chapitre 11

Conclusion et perspectives

11.1 Rappel de la problématique

Dans le cadre de Java, de nombreux outils ont été proposés afin de faire évoluer une application au cours du temps, mais également d'adapter la machine virtuelle Java aux contraintes d'exécution. Cependant les solutions existantes sont difficilement combinables entre elles car souvent *ad-hoc* à un contexte donné. De plus, elles peuvent se ramener dans la plupart des cas à des modifications de la machine virtuelle. En y étant intégrées et combinées, elles peuvent alléger la charge du programmeur, mais ne le sont pas, souvent pour des raisons de performances d'exécution. Cependant de nombreux travaux autour des plates-formes d'exécution dynamiquement adaptatives montrent que cette solution permet une meilleure gestion de l'exécution d'une application, de son évolution au cours du temps et de la configuration de son environnement d'exécution, par rapport aux solutions modifiants son code source ou compilé.

11.2 Notre solution : Corosol

Pour aboutir à cet objectif dans le cadre de Java, la machine virtuelle doit posséder une architecture dont les composants sont facilement identifiables pour être ensuite configurés et/ou modifiés. Elle doit également être suffisamment ouverte afin que d'autres composants puissent être ajoutés.

C'est dans ce contexte que nos travaux de thèse se déroulent. Ils proposent :

- un modèle formel qui représente le découpage d'une machine virtuelle en termes de composants, ainsi que leur façon de communiquer entre eux,

- un mécanisme de réflexivité *via* lequel n'importe qu'elle partie de l'architecture de cette machine virtuelle peut être facilement modifié avant son démarrage, mais également à la volée, directement à partir de l'application qu'elle exécute.

Ainsi, nous présentons une machine virtuelle Java, Corosol, entièrement écrite en Java, dont les composants sont facilement idenfiables et modifiables à la volée grâce à de meilleures capacités de réflexivité structurelle et comportementale en comparaisons avec les machines virtuelles classiques. Corosol possédant une architecture ouverte, l'application qu'elle exécute peut en modifier les composants pour évoluer ou adapter son exécution à un contexte donné.

Dans la première partie de cette thèse, le découpage en composants de la machine virtuelle Java se base sur la spécification de Sun. L'architecture qui en résulte est modulaire. Elle permet de paramétrer Corosol avant son démarrage, l'implantation de chaque composant pouvant être modifiée *via* un fichier de configuration.

Dans la seconde partie de cette thèse, les mécanismes réflexifs qui permettent à notre machine virtuelle d'interagir avec l'application qu'elle exécute sont mis en œuvre. L'application peut ainsi observer les différents composants de Corosol, les remplacer ou en intégrer d'autres. Le mécanisme de réflexivité est symbiotique : les deux couches, interprète et application, sont fusionnées pour n'en former qu'une. Les objets natifs Corosol, c'est-à-dire ces composants, cohabitent avec les objets créés par l'application exécutée. Ce mécanisme de réflexivité symbiotique est constitué de deux parties :

- le mécanisme *up* qui permet de faire apparaître les composants natifs de la machine virtuelle au niveau de l'application exécutée par celle-ci ; ces composants apparaissent sous la forme d'objets Java ;
- le mécanisme *down* qui permet de convertir la représentation des objets créés durant l'exécution de l'application et par celle-ci en objets natifs manipulables par Corosol.

La troisième partie de cette thèse illustre par des exemples les possibilités offertes par notre machine virtuelle. Nous avons montré étape par étape comment ajouter le support d'un nouveau type primitif avant exécution. Nous avons aussi illustré les possibilités offertes par la symbiose avec l'application exécutée : nous avons en particulier montré comment remplacer le tas de notre machine virtuelle ainsi que son ordonnanceur de *threads* à partir de l'application et pendant son exécution. En dernier exemple, cette troisième partie a montré comment s'effectue le remplacement du mécanisme de résolution de méthode de la spécification de Sun par celui des multi-méthodes.

11.2.1 Résultats obtenus

L'objectif initial, qui était d'obtenir une machine virtuelle Java ouverte, écrite en Java et modifiable statiquement et dynamiquement *via* l'application, a donc été atteint. Nous avons réussi à reproduire ce qui avait déjà été accompli dans d'autres langages de programmation comme Lisp ou SmallTalk.

Corosol peut maintenant servir de plate-forme de recherche en Java. C'est également le cas de JikesRVM [Jik]. Cependant, contrairement à cette machine virtuelle, notre approche n'est pas entièrement centrée sur la compilation, mais plutôt sur une architecture modulaire clairement définie couplée à de puissants mécanismes réflexifs et de génération de *bytecode*. Les performances sont moins bonnes que dans l'approche de JikesRVM, mais permettent plus de souplesses lorsqu'il s'agit de permettre la modification d'un ou plusieurs composants *lors de l'exécution*.

11.3 Perspectives

Si sa forte modularité et sa faculté d'être en symbiose avec l'application exécutée sont les points forts de Corosol, sa vitesse d'exécution est cependant son point faible. Nos mesures établissent des performances de l'ordre cinquante fois plus lent qu'une machine virtuelle Java écrite dans un langage comme C. Ces mauvaises performances sont dues :

- au double niveau d'interprétation, en grande majorité,
- à la lourdeur de certains mécanismes de Corosol, comme par exemple l'abstraction de la représentation des références, qui repose sur l'utilisation systématique des objets `JHeapObject` de la machine virtuelle *hôte* pour chacun de leur accès.

Nos travaux futurs consistent donc à améliorer les performances d'exécution sans pour autant porter atteinte à l'architecture que nous avons réalisée. Une possible voie est d'introduire et d'entremêler dans le *bytecode* de l'application à exécuter le *bytecode* de notre machine virtuelle Corosol. Il en résulterait un seul niveau d'interprétation, le *bytecode* résultat étant exécuté par une machine virtuelle native, écrite par exemple en C.

L'exemple de l'ajout des nombres complexes (chapitre 9) comme type primitif du langage de programmation Java nous a montré qu'il était aussi nécessaire d'avoir un compilateur collaborant étroitement avec notre machine virtuelle. Nous avons prévu une architecture de représentation du *bytecode* assez modulaire pour inventer et supporter n'importe quelle extension de *bytecode*. Le compilateur à réaliser pourrait ainsi utiliser les éléments de cette

architecture pour sa production de *bytecode*.

Nous désirons également améliorer le mécanisme *down* de la symbiose au sein de Corosol. En Smalltalk, son implantation est facilitée grâce à sa machine virtuelle entièrement décrite en Smalltalk mais surtout parce qu'elle fournie la possibilité de définir de nouvelles méta-classes, ce qui n'est pas le cas en Java. Pour l'instant, Corosol réalise le mécanisme *down* de la symbiose en générant des mandataires (voir section 8.3), car il n'est pas possible d'étendre les méta-classes de la machine virtuelle *hôte* fournies au sein du paquetage `java.lang.reflect`. Depuis sa version 5, Java fournit un paquetage d'instrumentation du *bytecode* (`java.lang.instrument`) dans lequel il est possible de recharger une classe au sein de la machine virtuelle durant l'exécution. Le *bytecode* des classes peut ainsi être enrichi avant d'y être recharger. Cependant, cette opération ne peut se réaliser sur les classes chargées par la machine virtuelle Java, lors de son démarrage. Les méta-classes du paquetage `java.lang.reflect` n'y font pas exception. La machine virtuelle *hôte* doit donc être plus ouverte et fournir des méta-classes pouvant être étendues à l'exécution et pas seulement consultées. Cela permettrait une réalisation plus simple du mécanisme *down* de la symbiose dans Corosol.

La symbiose de notre machine virtuelle avec l'application qu'elle exécute propose de nombreuses possibilités d'extensions de Java, de sa machine virtuelle et naturellement de l'application. Elle rend notre machine virtuelle ouverte et tous ses composants sont alors disponibles pour être simplement observés, utilisés mais surtout remplacés. Cette possibilité de remplacement issue de l'ouverture de notre machine virtuelle présente des problèmes de sécurité évidents que nous n'avons pas abordés dans cette thèse. En effet, un remplacement inadapté d'un composant peut ainsi mettre en péril le déroulement de l'exécution, mais aussi de la machine virtuelle elle-même. Nos travaux futurs visent aussi à établir une police de sécurité adaptée permettant un contrôle de l'accès aux composants de notre machine virtuelle et en particulier pour leur remplacement. Une question intéressante concernant cette police de sécurité sera a étudier : doit-elle être elle aussi modifiable à n'importe quel moment par l'application ?

Cette dernière question en soulève d'ailleurs une autre beaucoup plus générale et qui concerne le remplacement des composants de notre machine virtuelle : doit-on aussi règlementer les différents moments où ceux-ci sont modifiés ou remplacés ?

Annexes

Dans cette thèse, nous utilisons différents sortes de diagrammes pour illustrer des idées importantes :

- Les diagrammes de classes décrivant les classes, leur structures et les relations statiques entre elles.
- Les diagramme de séquences montrant le flot des requêtes entre les objets.

Les diagrammes de classes et d'objets sont basés sur OMT (Object Modeling Technique). Les diagrammes de séquences sont tirés de Objectory et de la méthode Booch.

Diagrammes de classes

La figure 11.1 illustre la notation OMT pour les classes abstraites et les classes concrêtes. Une classe abstraite (ou une interface au sens Java) est représentée par une boîte composée de deux parties :

- une partie supérieure, où figure le nom de classe en caractère gras,
- une partie inférieure, où apparaissent les méthodes clés de celles-ci en italique.

Une classe concrête est quant à elle représentée par une même boite, avec en plus une dernière partie inférieure située en dessous de celle où est décrite les méthodes. Elle se compose de la liste des variables d'instances de la classe. L'information sur le type de chacune d'entre elle est optionnel. Dans cette thèse, nous utilisons la convention C++, qui met le nom du type avant le nom d'une méthode (ce qui précise son type de retour), d'un de ses paramètres, ou encore d'une variable d'instance. Les méthodes d'une classe concrête n'apparaissent pas en italique, contrairement à la description d'une classe abstraite.

La notation OMT pour l'héritage entre les classes est un triangle qui relie une ou plusieurs sous-classes à la classe mère (voir figure 11.2).

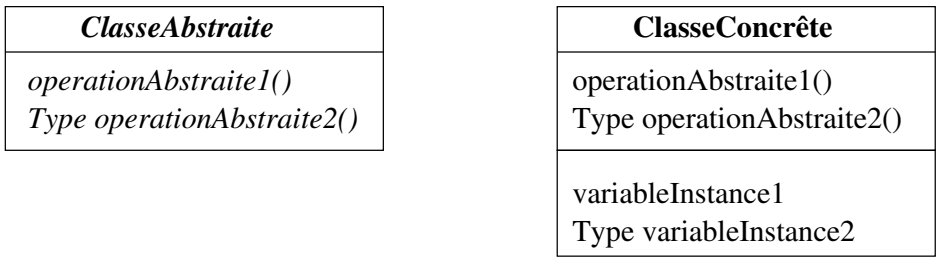

FIG. 11.1 – Classes abstraite et classe concrête.

La relation d'aggrégation entre classes est indiquée par une ligne fléchée possédant un diamant à la base. Cette flèche pointe vers la classe qui est aggrégée. Si elle ne possède pas de diamant à sa base, cette ligne fléchée exprime simplement l'accointance. Un nom peut apparaître à côté de cette même base pour préciser le nom d'une instance de classe aggrégée.

OMT définit aussi un cercle plein signifiant «plus qu'un». Lorsqu'un cerle apparait à la tête d'une référence, il signifie que de multiple objets sont en train d'être référencés ou agrégés.

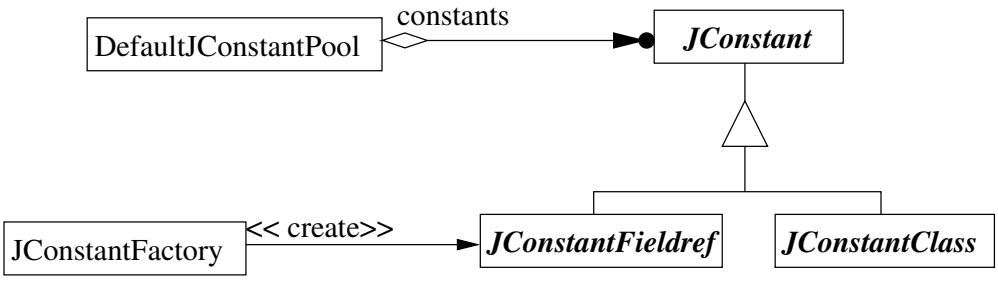

FIG. 11.2 – Relation entre classes

Diagrammes de séquences

Un diagramme de séquences (voir figure11.3) montre l'ordre dans lequel les requêtes entre les objets sont exécutées. Dans celui-ci, le temps s'écoule de haut en bas. Une ligne solide verticale indique la durée de vie d'un objet en particulier. La convention des noms pour les objets est la suivante : le nom de l'objet suivi d'un « : » puis du nom de sa classe. Si l'objet n'a pas encore été instancié depuis le début de la mesure du temps, sa ligne vertical apparait en pointillés jusqu'au point de sa création.

Un rectangle vertical signifie que cet objet est actif : il est alors en train de gérer une requête. Une méthode peut envoyer des requêtes vers d'autres objets. Ceux-ci sont indiqués avec un arc horizontal pointant vers l'objet receveur. Le nom de la requête est montré au dessus de cet arc. Une requête de création d'objet est montrée comme une line pointillée se terminant par un arc. Une requête de l'objet émetteur vers ce même objet émetteur pointe sur lui-même. Un arc horizontal pointant de l'objet receveur vers l'objet émetteur indique la fin de l'exécution d'une requête. Si une valeur est retournée, son nom est situé au-dessus de cet arc.

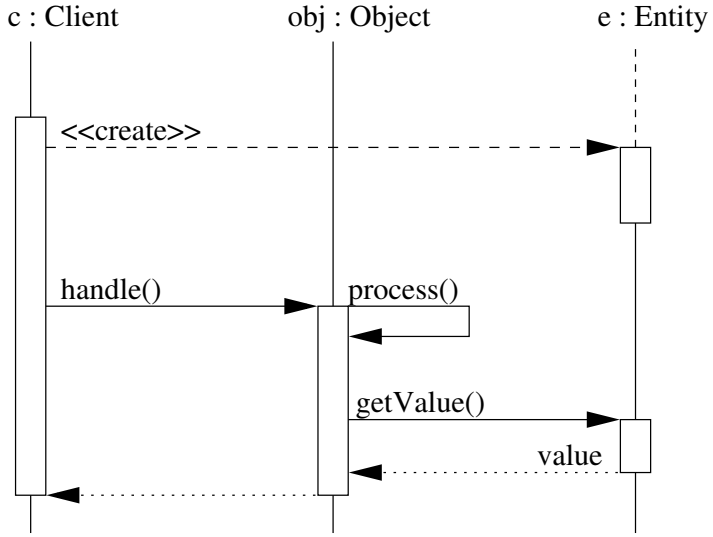

FIG. 11.3 – Exemple d'un diagramme de séquences.

Bibliographie

[Aa01] B. Alpern and al. Implementing Jalapeño in java. Technical report, IBM, 2001.

[Aa02] B. Antonsson and al. JRockit - The Faster Server JVM, 2002. White paper.

[AF94] Giuseppe Attardi and Tito Flagella. A customisable memory management framework. Technical Report TR-94-010, International Computer Science Institute, Berkeley, 1994. Also Proceedings of the USENIX C++ Conference, Cambridge, MA, 1994.

[AFI98] Giuseppe Attardi, Tito Flagella, and Pietro Iglio. A customisable memory management framework for C++. 28(11) :1143–1183, November 1998.

[BAS99a] Derek Lieber Mark Mergen Bowen Alpern, Anthony Cocchi and Vivek Sarkar. Jalapeño : a compiler-supported java virtual machine for servers. In *Workshop on Compiler Support for Software System (WCSSS 99)*, Atlanta, GA, May 1999.

[BAS99b] John J. Barton Anthony Cocchi Susan Flynn Hummel Derek Lieber-Mark Mergen Ton Ngo Janice Shepherd Bowen Alpern, Dick Attanasio and Stephen Smith. Implementing jalapeño in java. In *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '99)*, Denver, Colorado, November 1999.

[BAW00] J. J. Barton M. G. Burke P Cheng J.-D. Choi A. Cocchi S. J. Fink D. Grove M. Hind S. F. Hummel D. Lieber V. Litvinov M. F. Mergen T. Ngo J. R. Russell V. Sarkar M. J. Serrano J. C. Shepherd S. E. Smith V. C. Sreedhar H. Srinivasan B. Alpern, C. R. Attanasio and J. Whaley. The jalapeño virtual machine. *IBM System Journal*, 39(1), February 2000.

[BBD+00] Greg Bollela, Ben Brosgol, Peter Dibble, Steve Furr, James Gosling, David Hardin, and Mark Turnbull. *The Real-Time Specification for Java.* Addison-Wesley, 2000.

[BCS] Éric Bruneton, Thierry Coupaye, and Jean-Bernard Stefani. The fractal component model, version 2.0-3. Online documentation http ://fractal.objectweb.org/specification/.

[BCS02] Éric Bruneton, Thierry Coupaye, and Jean-Bernard Stefani. Recursive and dynamic software composition with sharing. In *WCOP'02*, Malaga, Spain, June 2002.

[BFGP02] David Basin, Stephan Friedruch, Marek Gawkowski, and Joachim Posegga. Bytecode model checking : An experimental analysis. In Dragan Bosnacki and Stefan Leue, editors, *LNCS 2318, Model Checking Software, 9th international SPIN Workshop*, pages 42–59. Springer, April 2002.

[BGW02] J. Brichau, K. Gybels, and R. Wuyts. Towards a linguistic symbiosis of an object-oriented and logic programming language, 2002.

[BH02] J. Baker and W. Hsieh. Runtime aspect weaving through metaprogramming. In *Proceedings of the 1st International conference on Aspect-Oriented Software Development*, 2002.

[BLC02] Éric Bruneton, Romain Lenglet, and Thierry Coupaye. ASM : a code manipulation tool to implement adaptable systems. In *Proceedings of the ASF (ACM SIGOPS France) Journées Composants 2002 : Systèmes à composants adaptables et extensibles (Adaptable and extensible component systems)*, November 2002.

[Boy02] Chandrasekhar Boyapati. Towards an extensible virtual machine. Technical report, MIT Laboratory for Computer Science, Cambridge, 2002.

[BS99] Noury Bouraqadi-Saadani. Java et la réflexion. Technical report, École des Mines de Nantes - Dépt. Informatique, 1999.

[BSL01] Noury M. N. Bouraqadi-Saadani and Thomas Ledoux. Le point sur la programmation par aspects. *Technique et science informatiques*, 20(4), 2001.

[CCK98] Geoff Cohen, Jeff Chase, and David Kaminsky. Automatic program transformation with JOIE. In *1998 USENIX Annual Technical Symposium*, pages 167–178, 1998.

[CCL99] R. Chignoli, P. Crescenzo, and Ph. Lahire. OFL : une machine virtuelle objet flexible pour la gestion d'objets persistants et mobiles. Rapport de Recherche RR-99-09, Laboratoire I3S (UNSA/CNRS), Sophia-Antipolis, France, mars 1999.

[Chi98] Shigeru Chiba. Javassist - A reflection-based programming wizard for java, October 02 1998.

[Chi00] Shigeru Chiba. Load-time structural reflection in java. In *ECOOP*, pages 313–336, 2000.

[COR] CORBA Component Model. http ://www.omg.org/technology/documents/formal/components.htm.

[CT98] S. Chiba and M. Tatsubori. Yet another java.lang.Class. In *Proceedings of the Workshop on Reflective Object-Oriented Programming and Systems at the 12th European Conference on Object-Oriented Programming (ECOOP'98)*, pages 372–373, Brussels, Belgium, July 1998.

[DA01a] P. Doyle and T. Abdelrahman. Jupiter : A modular and extensible JVM. In *Proceedings of the Third Annual Workshop on Java for High-Performance Computing, ACM International Conference on Supercomputing*, pages 37–48. ACM, June 2001.

[DA01b] Patrick Doyle and Tarek S. Abdelrahaman. Jupiter : A Modular and Extensible JVM. Technical report, University of Toronto, 2001.

[Dah01] M. Dahm. Byte code engineering with the bcel api. Technical report, Freie Universität Berlin, Institut für Informatik, april 2001.

[DBR03] Christophe Deleray, Nicolas Bedon, and Gilles Roussel. MyJVM : a 100% pure Java customizable Java Virtual Machine. In *Proceedings of the 2nd international conference on Principles and practice of programming in Java*, pages 177–179. Computer Science Press, Inc., 2003.

[DBRD04] Christophe Deleray, Nicolas Bedon, Gilles Roussel, and Etienne Duris. Corosol une JVM modulaire paramétrable à la volée. In Hermès, editor, *Langages et Modèles à 0bjets (LMO'04)*, volume 10 of *L'objet*, pages 89–102, Lille, France, March 2004. Revue des Sciences et Technologies de l'Information.

[DLS+01] Christopher Dutchyn, Paul Lu, Duane Szafron, Steven Bromling, and Wade Holst. Multi-dispatch in the java virtual machine : Design and implementation. In *COOTS*, pages 77–92, 2001.

[EJB] Entreprise Java Beans. Online documentation http ://java.sun.com/products/ejb.

[FBPS] Bertil Folliot, Carine Baillarguet, Ian Piumarta, and Lionel Seinturier. Aspects et Réflexivité pour la machine virtuelle virtuelle.

[FDR00] R. Forax, E. Duris, and G. Roussel. The Java MultiMethod Framework. In *TOOLS Pacific'00 Proceedings*, November 2000.

[FJ89] Brian Foote and Ralph E. Johnson. Reflective facilities in smalltalk-80. In Norman Meyrowitz, editor, *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, volume 24, pages 327–336, New York, NY, 1989. ACM Press.

[Fol00] Bertil Folliot. The virtual virtual machine project. In *IFIP Symposium on Computer Architecture and High Performance Computing*, Sao Paulo, Brazil, October 2000.

[For01] Rémi Forax. *Les multi-méthodes en Java*. PhD thesis, Université de Marne-la-Vallée, December 2001. 240 pages.

[FPR98] Bertil Folliot, Ian Piumarta, and Fabio Riccardi. A dynamically configurable, multi-language execution platform. In *8th ACM SIGOPS European Workshop*, pages 175–181, Sintra, Portugal, September 1998.

[GH01] E. Gagnon and L. Hendren. SableVM : A research framework for the efficient execution of Java bytecode. In *Proceedings of the Java Virtual Machine Research and Technology Symposium (JVM '01)*, pages 27–39, April 2001.

[GHJV99] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns, Catalogue de modèles de conception réutilisables*. Vuibert, 1999.

[GK96] M. Golm and J. Kleinder. MetaXa : An Efficient Run-Time Meta Architecture for Java. In *IWOOOS'96*, 1996.

[GK98] M. Golm and J. Kleinder. MetaXa and the Future of Reflection. In *OOPSLA Workshop on Reflective Programming in C++ and Java*, pages 1–5, 1998.

[GR83] Adele Goldberg and David Robson. *Smalltalk-80 : The Language and Its Implementation*. Addison-Wesley, 1983.

[GWDD06] Kris Gybels, Roel Wuyts, Stéphane Ducasse, and Maja D'Hondt. Inter-language reflection : A conceptual model and its implementation. *Elesevier International Journal on Computer Languages, Systems and Structures (to appear)*, to appear(to appear), 2006.

[Hot] Hotspot jvm. http ://java.sun.com/products/hotspot/.

[IKM+97] Dan Ingalls, Ted Kaehler, John Maloney, Scott Wallace, and Alan Kay. Back to the future : The story of Squeak, A practical Smalltalk written in itself. In *Proceedings OOPSLA '97*, pages 318–326. ACM Press, November 1997.

[IMY92] Yuuji Ichisugi, Satoshi Matsuoka, and Akinori Yonezawa. RbCl : A reflective object-oriented concurrent language without a run-time kernel. In *IMSA'92*, pages 24–35, 1992.

[Jav] Java Card Technology. http ://java.sun.com/products/javacard/.

[Jik] Jikes RVM. http ://www-124.ibm.com/developerworks/oss/ jikesrvm/.

[KHH+01] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. Getting started with AspectJ. *Communications of the ACM*, 44(10) :59–65, 2001.

[LG97] Han Bok Lee and Benjamin G.Zorn. BIT : A Tool for Instrumenting Java Bytecodes. In *USENIX Symposium on Internet Technologies and Systems Proceedings*, pages 73–82, Monterey, California, december 1997. USENIX.

[LMG00] Brian T. Lewis, Bernd Mathiske, and Neal M. Gafter. Architecture of the pevm : A high-performance orthogonally persistent java virtual machine. In *POS*, pages 18–33, 2000.

[LQP92] Bernard Lang, Christian Queinnec, and José Piquer. Garbage collecting the world. pages 39–50, 1992.

[LY99] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. SUN Press - Addison-Wesley, 2 edition, 1999.

[Mae87] Pattie Maes. *Computational Reflection*. PhD thesis, Vrije Universiteit Brussel, 1987.

[Meu98] Wolfgang De Meuter. Agora : The story of the simplest MOP in the world - or - The Scheme of Object-Orientation. In A. Taivalsaari J. Noble, I. Moore, editor, *Prototype-based Programming*. Springer-Verlag, 1998.

[Mic97] Golm Michael. Design and Implementation of a Meta Architecture for Java. Master's thesis, University of Erlangen-Nürnbergs, Departement of Computer Science IV, January 1997.

[MJD96] Jacques Malenfant, Marco Jacques, and François-Nicolas Demers. A tutorial on behavioral reflection and its implementation. In Gregor Kiczales, editor, *Proceedings of the Reflection 96 Conference*, pages 1–20, California, April 1996.

[NET] .NET/COM+/(DCOM). Online documentation http ://msdn.microsoft.com/net.

[OB98] A. Olivia and L. E. Buzato. Composition of meta-objects in Guaraná. In *Proceedings of the 13th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'98)*, pages 82–86, Vancouver, Canada, October 1998.

[OB99] A. Olivia and L. E. Buzato. The design and implementation of Guaraná. In *Proceedings of the 5th USENIX Conference on Object-Oriented Technologies and Systems (COOTS'99)*, pages 203–216, San Diego, California, USA, May 1999.

[OGB98] A. Olivia, I. C. Garcia, and L. E. Buzato. The reflective architecture of Guaraná. Technical Report IC-98-14, Universidade Estadual de Campinas, Brazil, April 1998.

[OTGF04] F. Ogel, G. Thomas, A. Galland, and B. Folliot. MVV : une plate-forme à composants dynamiquement reconfigurables. 23(10) :1269–1299, 2004.

[PDFS01] R. Pawlak, L. Duchien, G. Florin, and L. Seinturier. Dynamic wrappers : Handling the composition issue with JAC. In *TOOLS 2001*, 2001.

[PFSB00] Ian Piumarta, Bertil Folliot, Lionel Seinturier, and Carine Baillarguet. Highly configurable operating systems : the VVM approach. In *ECOOP'2000 Workshop on Object Orientation and Operating Systems*, Cannes, France, June 2000.

[Riv96] Fred Rivard. Smalltalk : a reflective language. Technical report, Laboratoire Jules Verne, Ecole des Mines de Nantes & Object Technology International Inc., France, October 1996.

[RMI] Java Remote Method Invocation Specification. http ://java.sun.com/j2se/1.4.2/docs/guide/rmi/spec/rmiTOC.html.

[SBN+97] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser : A dynamic data race detector for multi-threaded programs. In *16th ACM Symposium on Operating Systems Principles*. ACM, 1997.

[SMCS04] S. Masoud Sadjadi, Philip K. McKinley, Betty H. C. Cheng, and R. E. Kurt Stirewalt. TRAP/J : Transparent Generation of Adaptable Java Programs. *Lecture Notes in Computer Science*, 3291 :1243–1261, 2004.

[Smi82] Brian Cantwell Smith. *Procedural reflection in programming languages*. PhD thesis, Massachussets Institute of Technology, February 1982. 762 pages.

[Smi84] Brian Cantwell Smith. Reflection and semantics in LISP. In *Proceedings of the 11th symposium on Principles of programming languages*, pages 23–35, 1984.

[SP] D'Hondt Theo De Hondt Koen Lucas Carine Van Limberghen Marc Steyaert Patrick, Codenie Wim. Nested mixin-

methods in agora. In *Proceedings of the 7th European Conference on Object-Oriented Programming, number 707 in Lecture Notes in Computer Science*, pages 197–219. Springer-Verlag.

[Spi97] The SPIN Model Checker. In *IEEE Trans. on Software Engineering*, volume 23, pages 179–295, May 1997.

[Ste94] P. Steyaert. *Open design for object-oriented languages, a foundation for specialisable reflective language framework.* PhD thesis, Brussels Free University, 1994.

[Szy98] Clemens Szyperski. *Component Software.* ISBN : 0-201-17888-5. Addison-Wesley, 1998.

[Tai98] Antero Taivalsaari. Implementing a Java virtual machine in the Java programming language. Technical report, March 1998.

[TC98] Michiaki Tatsubori and Shigeru Chiba. Programming support of design patterns with compile-time reflection. In *Proceedings of OOPSLA'98 Workshop on Reflective Programming in C++ and Java*, pages 56–60, 1998.

[TCIK99] Michiaki Tatsubori, Shigeru Chiba, Kozo Itano, and Marc-Olivier Killijian. OpenJava : A class-based macro system for java. In *OORaSE*, pages 117–133, 1999.

[THL01] P. Tullmann, M. Hibler, and J. Lepreau. Janos : a Java-oriented OS for Active Networks. *IEEE Jour. on Selected Areas in Com.*, 19(3) :501–510, March 2001.

[Tja99] S. J. Tjasink. PLAVA : a persistent, lightweight Java Virtual Machine. Master's thesis, University of Cape Town, Faculty of Sciences, Departement of Computer Science, February 1999.

[VEBO01] A. J. Van Engen, M. K. Bradshaw, and N. Oostendorp. Extending Java to support shared resource protection and deadlock detection in threads programming. *ACM Crossroads*, 2001. http ://www.acm.org/crossroads/xrds4-2/dynac.html.

[Ven98] Bill Venners. *Inside the Java Virtual Machine.* The Java Masters Series. Computing McGraw-Hill, February 1998. Chapter 9 : Garbage Collection.

[WD01] Roel Wuyts and Stéphane Ducasse. Symbiotic reflection between an object-oriented and a logic programming language. In *In ECOOP 2001 International Workshop on MultiParadigm Programming with Object-Oriented Languages*, 2001.

[Wil92] Paul R. Wilson. Uniprocessor garbage collection techniques. In *Proc. Int. Workshop on Memory Management*, number 637, Saint-Malo (France), 1992. Springer-Verlag.

[Zim96] Chris Zimmermann. *Advances in Object-Oriented Metalevel Architectures and Reflection*. CRC Press, Inc., Boca Raton, FL, USA, 1996.

[ÉT04] Éric Tanter. *From Metaobject Protocols to Versatile Kernels for Aspect-Oriented Programming*. PhD thesis, University of Nantes, France, and University of Chile, Chile, nov 2004.

Index